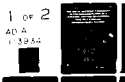
AD-A113 934 PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING F/G 5/8
THE USE OF DATABASE TECHNIQUES IN THE IMPLEMENTATION OF A SYNTA--ETC(U)
DEC 81 E C SEED, H J SIEGEL AFOSR-78-3581
UNCLASSIFIED TR-EE-81-49 AFOSR-TR-82-0301 NL



THE USE OF DATABASE TECHNIQUES IN THE IMPLEMENTATION OF A SYNTACTIC PATTERN RECOGNITION TASK ON A PARALLEL RECONFIGURABLE MACHINE

Elizabeth Cathro Seed
Howard Jay Siegel

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 81-49
December 1981

This work was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581.

THE USE OF DATABASE TECHNIQUES IN THE IMPLEMENTATION OF A SYNTACTIC PATTERN RECOGNITION TASK ON A PARALLEL RECONFIGURABLE MACHINE

Elizabeth Cathro Seed

Howard Jay Siegel

School of Electrical Engineering

Purdue University

West Lafayette, Indiana 47907

Purdue University

TR-EE 81-49

December 1981

This work was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581.

ACKNOWLEDGEMENTS

The authors wish to thank the following people for their comments on the contents of this work: Professor Rangasami L. Kashyap, Professor King-Sun Fu, and William C. Karashin. The authors also thank Mickey Krebs for typing the manuscript.

This report is based on the Masters Thesis of Elizabeth Cathro Seed. The research was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The use of syntactic pattern recognition has been shown to be an effective technique for picture processing. Syntactic pattern recognition is, however, computationally time-consuming. The way in which a parallel SIMD/MIMD machine, PASM, can be used to decrease the processing time of these tasks is examined.

Parallel machines have been used predominantly for decreasing the processing time of numerical problems in which the data is frequently well-ordered. In contrast, a syntactic pattern recognition task would use a parallel machine to perform multiple search, comparison, and string manipulator operations on some relatively complex data structures.

A solution to the problem of implementing a specific parallel syntactic pattern recognition task, a parallel tree automaton, through the use of a relational database and relational language is proposed. The use of a CODASYL database and database language is also investigated.

Two algorithms for implementing the parallel tree automaton are described. The problem of obtaining a reasonable processor and data allocation scheme for the two algorithms and for the two relational programs derived from the two algorithms is discussed. A comparison of the different problems posed by each algorithm is made.

CHAPTER 1

INTRODUCTION

Syntactic pattern recognition tasks as performed on serial processors are time-consuming. The execution time of the syntactic pattern recognition task may be reduced through the use of a parallel picture processing machine.

The implementation of two algorithms for a tree automaton, a syntactic pattern recognition task, will be considered in this paper. The two algorithms are closely related versions of a minimum-distance structure preserved error correcting tree automaton (minimum-distance SPECTA). The machine being considered for the implementation is PASM, a partitionable SIMD/MIMD machine.

An introduction to machine configurations and PASM is given in Chapter 2. A brief introduction to tree grammars and the use of tree grammars in picture processing is given in Chapter 3. The two algorithms for parallel tree automaton are presented in Chapter 5.

The algorithms will be implemented using databases and database languages. An introduction to the three database approaches, the relational approach, the hierarchical approach, and the network approach, is given in Chapter 4.

Chapter 5 presents and describes the algorithms. A discussion of some factors that should be considered for a simulation of the

algorithms on PASM is also included in Chapter 5 as is a discussion of some processor and data allocation schemes for both algorithms.

The partial translation of the algorithms into PASM pseudo programs using a network database system is described in Chapter 6. Chapter 6 also contains the complete translation of the algorithms into illustrative pseudo programs using a relational language and some of PASM's proposed parallel language commands.

A simple suggestion for a processor and data allocation scheme for the relational programs, a block distribution of data according to the primary sort key, is given in Chapter 7. A brief discussion of some of the compilation problems posed by this illustrative allocation scheme is also included in Chapter 7.

CHAPTER 2

INTRODUCTION TO PASM

2.1 Introduction

As a result of the microprocessor revolution, it is now feasible to build a dynamically reconfigurable large-scale multimicroprocessor system capable of performing image processing and syntactic pattern recognition tasks more rapidly than previously possible. There are several types of parallel processing systems: SIMD, MSIMD, MIMD, and PSM.

An SIMD (single instruction stream - multiple data stream) machine [Fly] typically consists of a set of N processors, N memories, an interconnection network, and a control unit (e.g. Illiac IV [Bou]). The control unit broadcasts instructions to the processors and all active ("turned on") processors execute the same instruction at the same time. Each processor executes instructions using data taken from a memory to which only it is connected. The interconnection network allows interprocessor communication. An MSIMD (multiple-SIMD) system is a parallel processing system which can be structured as two or more independent SIMD machines (e.g. MAP [Nut]). An MIMD (multiple instruction stream - multiple data stream) machine [Fly] typically consists of N processors and N memories, where each processor may follow an independent instruction stream (e.g. C.mmp [WuB]). As with SIMD

architectures, there is a multiple data stream and an interconnection network. A PSM (partitionable SIMD/MIMD) system is a parallel processing system which can be structured as one or more independent SIMD and/or MIMD machines (e.g. PASM [SMS,SSK]).

2.2 Overview of PASM

A block diagram of PASM (a partitionable SIMD/MIMD system) is shown in Figure 2.1. The heart of the system is the Parallel Computation Unit (PCU), which contains N processors, N memory modules, and the interconnection network. The PCU processors are microprocessors that perform the actual SIMD and MIMD computations. The PCU memory modules are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The interconnection network provides a means of communication among the PCU processors and memory modules.

The Micro Controllers (MCs) are a set of microprocessors which act as the control units for the PCU processors in SIMD mode and orchestrate the activities of the PCU processors in MIMD mode. Control Storage (CS) contains the programs for the Micro Controllers. The Memory Management System (MMS) controls the loading and unloading of the PCU memory modules. The Memory Storage System (MSS) stores these files. The System Control Unit (SCU) is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM.

The processors, memory modules, and interconnection network of the PCU are organized as shown in Figure 2.2. A pair of memory units is used for each PCU memory module so that data can be moved between one

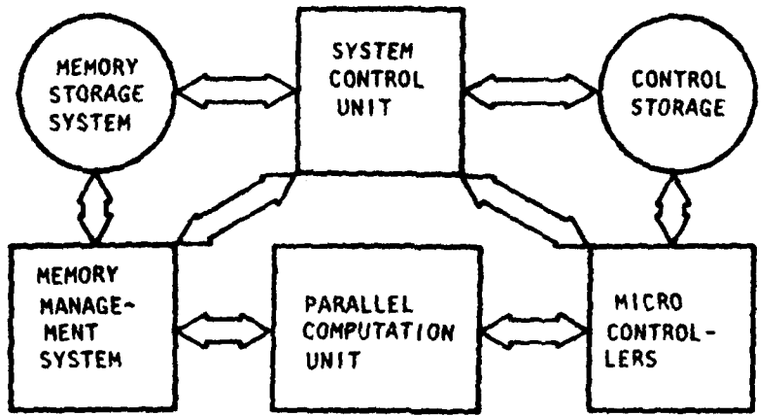


Figure 2.1: Block diagram overview of PASM.

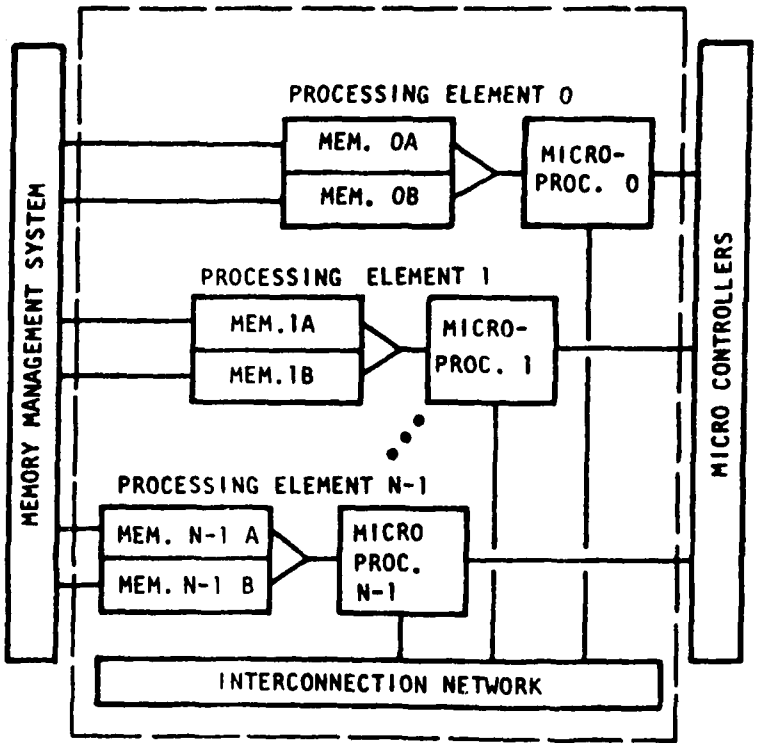


Figure 2.2: PE-to-PE configuration of the Parallel Computation Unit.

memory unit and the MSS while the PCU processor operates on data in the other memory unit. The processors, which are physically numbered (addressed) from 0 to N-1, where $N=2^n$, communicate through the interconnection network.

Many computations can be more efficiently executed if the N PCU processors are partitioned into many smaller groups of processors, each group behaving like an SIMD or an MIMD machine. The two interconnection networks being considered for PASM are the generalized cube [SiS,SmS,SM2] and the augmented data manipulator [SiS,SmS,SM1]. Both of these networks consist of n stages of switches and are controlled by routing tags. Both networks can be partitioned into independent subnetworks if all of the processing elements in a partition of the size $P = 2^p$ have the same value in the low-order n-p bit positions of their addresses.

The method to provide multiple controllers for forming virtual independent machines is shown in Figure 2.3. There are $Q=2^q$ MCs, physically addressed (numbered) from 0 to Q-1. Each MC controls N/Q PCU processors. There is an MC memory module for each MC. Each MC memory module contains a pair of memories so that memory loading and computations can be overlapped. A virtual SIMD machine of size RN/Q, where $R=2^r$ and $1 \leq r \leq q$, is obtained by loading R MC memory modules with the same instructions simultaneously. Similarly, a virtual MIMD machine of size RN/Q is obtained by combining the efforts of the PCU processors of R MCs. For either SIMD or MIMD mode, the physical addresses of these R MCs must have the same low-order q-r bits since the physical addresses of all PCU processors in a partition must agree in

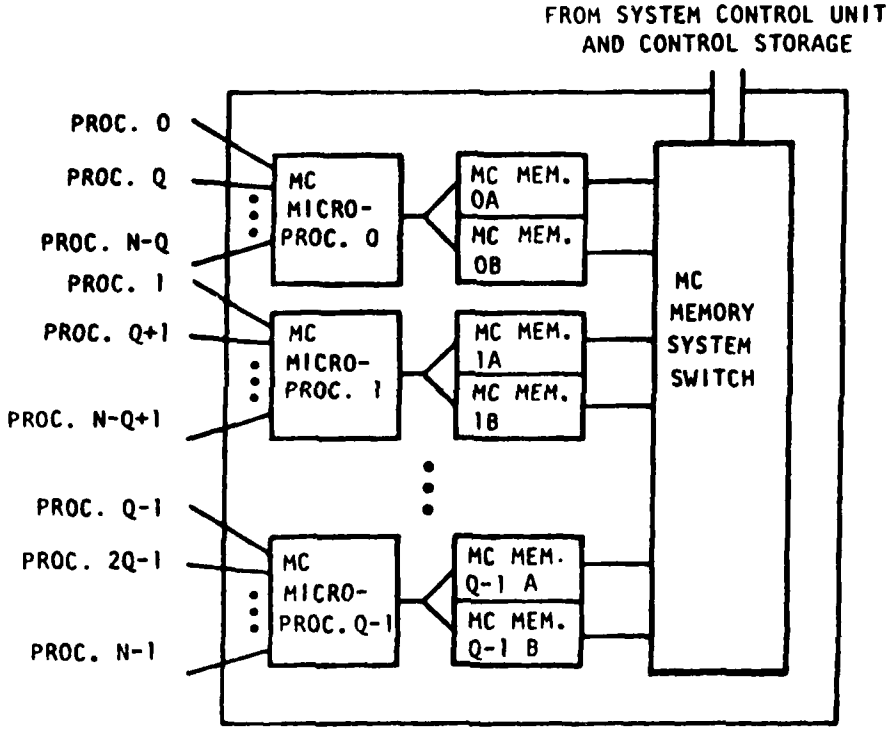


Figure 2.3: PASM Micro Controllers.

their low-order bits in order for the interconnection network to function properly. Possible values for N and Q are 1024 and 16, respectively.

This brief overview was provided as background for the following chapters. More details about PASM and parallel machine interconnection networks can be found in [Si1-3,SiS,SmS,SM1,SM2,SMS,SSK].

CHAPTER 3

A BRIEF INTRODUCTION TO TREE GRAMMARS

3.1 Introduction

For the purposes of this paper, only a basic definition of a tree grammar is needed. This chapter is based on material in [CF1,CF2]. More detailed definitions of trees, tree grammars, and tree automata can be found in [LF1,LF2,Fu1,Fu2,FuB,CF1,CF2].

Definition 3.1. A grammar $G_i = (V,r,P,S)$ over $<\Sigma,r>$, a ranked alphabet, is a tree grammar in expansive form where

V is a set of terminal and nonterminal symbols,

Σ is a set of terminal symbols,

$r:\Sigma \rightarrow N$, where N is the set of non-negative integers, is the rank associated with symbols in Σ,

S is the starting symbol, and

P is a set of production rules of the form

$$X_0 \rightarrow \overset{x}{\diagdown\ \diagup\ \cdots\ \diagdown}\ X_1\ X_2\ \cdots\ X_{r(x)} \qquad \text{or} \qquad X_0 \rightarrow x$$

where $x \in \Sigma$ and $X_0, X_1, \ldots, X_{r(x)} \in V-\Sigma$ (the set of nonterminal symbols).

The term rank, as defined above, refers to the number of nonterminal symbols associated with the terminal symbols. At later

points in this paper there are references to the rank of a grammar rule (production rule) and the rank of a node in a tree. In these instances, the term rank refers to the number of nonterminal symbols associated with the terminal symbol of a particular grammar and the number of children of the specified node, respectively. The terminal symbol of a node is also often referred to as the node label.

3.2 A Tree Grammar for Picture Processing

When designing a tree grammar to describe patterns in a two-dimensional digitized picture, several factors need to be considered. Among these are the extraction of the tree structure from the picture and the choice of pattern primitives.

Once the digitized picture is windowed, a tree structure can be easily obtained from a window. If the pattern primitives chosen represent the gray levels, then each pixel in a window corresponds to a pattern primitive in the tree. Every pixel in a window is, therefore, a node in the tree, with the gray level of the pixel represented by the node label. Binary pattern primitives were chosen for simplicity such that a dark pixel corresponds to a 1, otherwise the pixel corresponds to a 0.

An example of the chosen tree structure for a 5x5 window is given in Figure 3.1. Using this tree structure and the previously defined binary pattern primitives, the 5x5 pattern in Figure 3.2 is represented by the tree in Figure 3.3. The tree structure chosen has the benefits of simplicity and symmetry. Less processing time is needed to extract a simple tree structure, and the symmetry helps to maximize the amount of

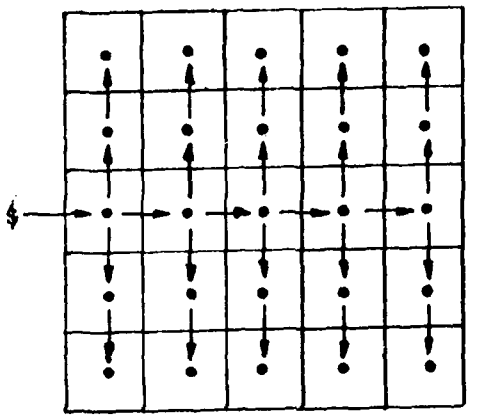


Figure 3.1: Tree structure [CF1,CF2].

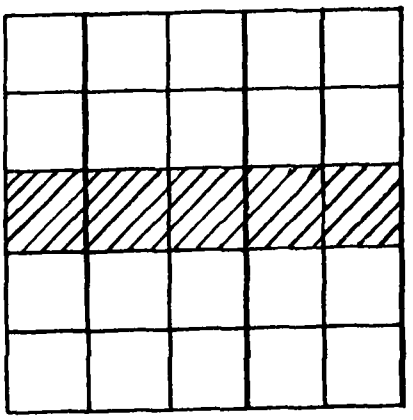


Figure 3.2: A Pattern [CF1,CF2].

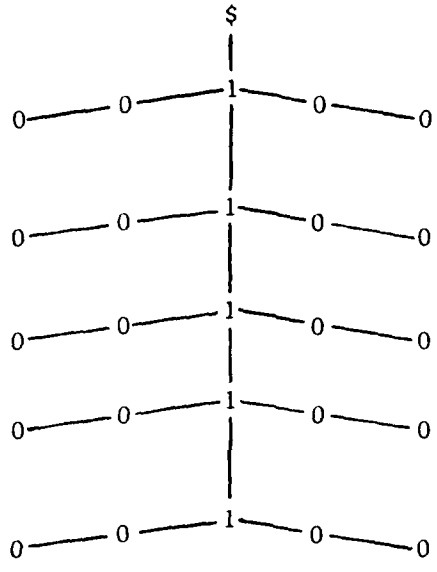


Figure 3.3: Tree representation [CF1,CF2].

parallel processing performed, thereby decreasing the overall processing time.

Having chosen the above tree structure and binary primitives, Chang and Fu [CF1,CF2] developed the following sample tree grammar G_0.

Example 3.1 Tree grammar G_0 generates many different patterns. For a wxw window it generates the following four patterns:

1. Only the pixels in the third row are 1's (Figure 3.2),
2. Only the pixels in the second row are 1's,
3. All the pixels are 1's,
4. All the pixels are 0's.

$G_0 = (V,r,P,S)$ over $\langle \Sigma, r \rangle$ where

$V = \{U_1, U_0, A_1, A_0, X_1, S, \$, 0, 1\}$

$\Sigma = \{$ ▨ $_1$, □ $_0$, $\$\}$

$r = \{0,1,2,3\}$

P : $S \rightarrow$ $\$_{U_1}$ (1); $\$_{U_0}$ (2); $\$_{A_1}$ (3); $\$_{A_0}$ (4)

$U_1 \rightarrow$ $1_{A_0 U_1 A_0}$ (5); $1_{A_0\ A_0}$ (6)

$U_0 \rightarrow$ $0_{A_0 U_0 X_1}$ (7); $0_{A_0\ X_1}$ (8)

$A_1 \rightarrow$ $1_{A_1 A_1 A_1}$ (9) $1_{A_1\ A_1}$ (10); 1_{A_1} (11); 1 (12)

$A_0 \rightarrow$ $0_{A_0 A_0 A_0}$ (13); $0_{A_0\ A_0}$ (14); 0_{A_0} (15); 0 (16)

$$X_1 \rightarrow \begin{matrix} 1 \\ | \\ A_0 \end{matrix} \quad (17); \qquad 1 \quad (18)$$

The above grammar is a first level tree grammar. It should be noted that a multilevel tree system may be used to process a picture. In a multilevel system, the tree grammars on the higher levels use the identified patterns on the lower levels as pattern primitives. These pattern primitives are then connected in a tree structure, as was done on the first level. In the multilevel system, the same tree automaton may be used to parse the tree structures on all levels. The multilevel system is an extension of a single level system.

CHAPTER 4

INTRODUCTION TO DATABASE

A simple definition of database is given in Martin [Mar].

> "A database is a collection of interrelated data stored together with controlled redundancy to serve one or more applications in an optimal fashion; the data are stored so that they are independent of programs which use the data; a common and controlled approach is used in adding new data and modifying and retrieving existing data within the database."

The concept of database to be applied in this paper is more specific. For the purposes of this paper, a database provides an organizational framework for the representation of data and relationships between data. The database technology also provides a series of data sublanguages: languages to manipulate data in a database environment.

Currently there are three accepted database models: the relational model, the hierarchical model, and the network or CODASYL (Conference on Data System Language) [HaW] model. Each model supports different data structures and each has a variety of associated data sublanguages. The data structures supported by the hierarchical model are a subset of

those supported by the network model. The hierarchical model uses tree structures whereas the network model is capable of using a variety of more complex graph structures.

The relational model uses relations or tables as its organizational structure. A relation (table) contains tuples (rows) and attributes (columns). Henceforth the terms used will be relation, tuple, and attribute. Associations between data items are indicated by grouping attributes, columns of data items, together to form a relation.

In Figure 4.1, the data in four records are logically related in a simple tree structure. The conceptual models of this simple tree structure for each database model are illustrated in Figures 4.2a,b,c.

For this simple example the network and hierarchical models yield similar conceptual structures. The one difference between the models is the added access paths from the node containing record A to the nodes containing records C and D in the network model, Figure 4.2c. This allows for significantly greater flexibility in the physical access of information from the network model. These access paths are not contained in the hierarchical model in Figure 4.2b because a node can only have one parent in a strict hierarchical database. In addition, a hierarchical database can be entered only through the root node and a child cannot access its parent unless the parent has already been accessed through the root node. The process of accessing a parent in a hierarchical model is a complex task.

The flexibility of the network model is increased since the added access paths allow bottom-up traversals, whereas the hierarchical model is restricted to top-down traversals through the root node. It should

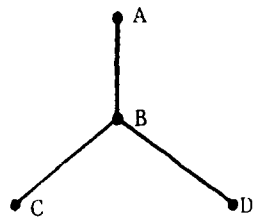


Figure 4.1: A Simple Tree Structure.

NODE	CHILD	PARENT
A	B	∅
B	C	A
B	D	A
C	∅	B
D	∅	B

Figure 4.2a: Relational model.

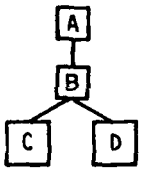


Figure 4.2b: Hierarchical model.

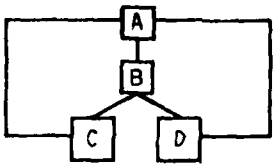


Figure 4.2c: Network model.

be noted that the hierarchical model need not be restricted to top-down traversal if extensions are made to the strict model. For example, IBM's hierarchical database system, IMS (Information Management System) [Dat,IC1], has secondary indexing to enable it to perform bottom-up traversals. It should also be noted that this facility is not always easily used and it can become quite space and time consuming.

In the above discussion, the access of the tree information in the hierarchical and network models is constrained by the physical database. The databases were structured to reflect the structure of the data items to be stored in the databases. The relational database, however, physically resembles a sequential file. Any access patterns must be constructed using the tuples and attributes in a relation. In Figure 4.2a, with the given tuples, the tree data represented in the relation may be logically accessed in a top-down fashion. However, additional tuples relating records C and D to record A and vice versa may be added thereby allowing a logical top-down or bottom-up traversal of the tree data.

Associated with each database model are data sublanguages: the subsets of languages concerned with operations on a database [Dat]. The commands of the data sublanguages usually take the form of calls to subroutines from a standard higher level programming language such as PL/1 or FORTRAN.

Some examples of these data sublanguages are DL/1 (Data Language/1) [Dat], used with the hierarchical IMS database system and the DML (Data Manipulation Language) used with GPLAN (Generalized Planning System) [HaW], a network database system. Most database systems also have stand

alone database languages such as IQF (Interactive Query Facility) [Dat,IC2], used with IMS, DBLOOK, used with the network database system SEED (Self Explaining Extended Database Management System) [Ger], and Query by Example, a relational language.

Some of the proposed languages for relational databases are either conventional data sublanguages or stand alone database languages, but not both. SEQUEL (Structured English Query Language) [Dat] is a relational language that can be used as a stand alone language, but it also is a data sublanguage because it can be embedded in a host language program.

In the instance where a host language is used, the host language provides the commands for performing arithmetic operations on the data obtained from the database using the database language. Database languages are generally restricted to providing the commands for nonarithmetic manipulation of the data in the database: searching and retrieval, updating, insertion, and deletion. One of the disadvantages of stand alone database languages is their inability to allow the user to perform calculations.

The database languages referred to in this paper will be DML and SEQUEL. They will be used as data sublanguages; the database commands will be imbedded in a program written using parallel language commands proposed for use on PASM [SSK]. A synopsis of the functions of the database commands used may be found in [HaW,Dat].

CHAPTER 5

TWO PARALLEL TREE PARSING ALGORITHMS

5.1 Introduction

A parallel tree automaton is designed to recognize those structures in a digital picture that are described by and, therefore, can be generated by a tree grammar. The first algorithm presented in this chapter is an implementation of a minimum-distance SPECTA. A minimum-distance SPECTA will find the tree among all trees that can be generated by a particular tree grammar, G, that has the same structure as a given input tree, B, which is a minimum distance from B. The distance between two trees, B and C, with the same structure is the number of substitution transformations required to derive tree C from tree B. A substitution transformation is the replacement of one node label (terminal symbol) in a tree by another. In terms of this algorithm, a minimum-distance SPECTA finds the tree among all trees that can be generated by the tree grammar, G, with the same structure as the input tree, B, with the minimum number of substitution errors. Note that a substitution error results from a substitution transformation. This will be explained in greater detail in the first subsection.

This algorithm will be referred to here as the leaf traversal algorithm. Following the first subsection is a subsection discussing

some processor and data allocation schemes for the leaf traversal algorithm.

The leaf traversal algorithm is the original tree parsing algorithm [CF1,CF2]. A slightly modified version of this algorithm, here referred to as the level traversal algorithm, is described in the third subsection. A discussion of some processor and data allocation schemes for this algorithm follows in the fourth subsection. A brief comparison of the two algorithms and the processor and data allocation schemes suggested for the algorithms is given in the last subsection.

5.2 The Leaf Traversal

5.2.1 The Algorithm

In Chang and Fu's algorithm [CF1,CF2], a leaf traversal of the tree is used. In the leaf traversal, all the leaves are processed in parallel. The processed leaves are then eliminated from the logical tree structure, and the leaves of the resulting tree are processed in parallel. The processing continues until the entire tree has been parsed. This approach has a major limitation: once the tree has been pruned to the "trunk," the process becomes serial with only one leaf being processed at a time.

The original specifications for the algorithm presented in this subsection can be found in [CF1,CF2]. The original algorithm has been rewritten here using a parallel language notation. The parallel language notation has been made as general as possible to assure the machine independence of the algorithm. The rewritten algorithm is presented in this subsection in Figure 5.1a and Figure 5.1b. Some of

Algorithm 1. A parallel parsing algorithm for the implementation of a minimum-distance SPECTA on a parallel machine.

```
Input: G=(V,r,P,S), the tree grammar, and an input tree B.
Output: Tree transition table for tree B.

for all node_i where node_i ε {i.f. nodes (leaves)} of tree B do
begin
   for all grammar rules with rank = 0 do
   begin

      if the terminal symbol of rule k = label of i.f. node_i
         then add triplet (X,0,k) to triplet table of i.f. node_i
         else add triplet (X,1,k) to triplet table of i.f. node_i

         /* X is the left-hand side nonterminal of the k^th */
         /* grammar rule                                    */

   end

   if the triplet table of i.f. node_i is nonempty
      then if more than one triplet has the same state
      /* The state is the first element of the triplet */
      /* the left-hand side nonterminal.               */
         then delete the triplet(s) with the larger number of errors

   frontier node_i = parent node of i.f. node_i

end
{l.l.f. nodes} = {frontier nodes with all children processed}
```

Figure 5.1a: The leaf traversal algorithm Part 1.

```
while the root node has not been processed
begin

   for all node_l where node_l ε {l.l.f. nodes} of tree B do
   begin
      n_l = rank of l.l.f. node_l
      for all different combinations of nonterminals of l.l.f.
            node_l's children's triplets do

      /* Combinations are formed by taking the nonterminal of  */
      /* one triplet from each child's triplet table.          */

      begin

         for all grammar rules with rank = n_l do
         begin

            if the nonterminal(s) of rule k = the nonterminal(s) of the
               triplet combination of l.l.f. node_l
               then if the terminal symbol of rule k = the label of the
                       l.l.f. node_l
                  then add (X,e_1+...+e_n_l,k) to triplet table of that node

                  else add (X,e_1+...+e_n_l+1,k) to triplet table of that
                       node
         end
      end

      if the triplet table of l.l.f. node_l is nonempty
          then if more than one triplet has the same state
             then delete the triplet(s) with the larger number of errors

       frontier node_l = parent node of l.l.f. node_l

   end

       {l.l.f. nodes} = {frontier nodes with all children processed}

end

if triplet (S,e,k) is in the triplet table of the root node
     then tree B is accepted with e errors
     else tree B is rejected
```

Figure 5.1b: The leaf traversal algorithm Part 2.

the terminology used in the algorithm is described in the following paragraphs.

Frequently the algorithm refers to specific types of "frontiers." A frontier is a specific type of node on the tree. It is either a leaf node on the original tree or a node with at least one processed child on the pruned tree. Frontier nodes that are leaves on the original tree are referred to as initial frontier nodes. Frontier nodes that are leaves on the pruned tree are referred to as lowest level frontier nodes. A leaf on the pruned tree is a node with all of its children processed. The initial frontier nodes and the lowest level frontier nodes are referred to as i.f. nodes and l.l.f. nodes, respectively.

The for all statements in the algorithm are used to indicate which data items may be processed in parallel. For example, for all i.f. nodes do, indicates that the operations within the do loop must be performed on all the i.f. nodes, that each i.f. node performs the operations independently, and, therefore, that the i.f. nodes may be processed in parallel.

It should be noted that the tree structure forces the processing of some nodes before others in the algorithm. As a result, it is impossible to parse the whole tree in parallel in one step. A number of serially executed parallel steps will be required. The number of serial steps, T, for a WxW window is W.

$$T = W + \frac{W}{2}.$$

For a 5x5 window, T is eight. The order of the processing of the nodes

for a 5x5 window for the leaf traversal algorithm is diagrammed in Figure 5.2.

The result of the processing of the nodes is a set of triplets for each node. Consider an arbitrary initial frontier node or lowest level frontier node, b. Each triplet is of the form (X,e,k), where X is the left-hand side nonterminal of the k^{th} grammar rule applied at node b, and e is the total number of substitution errors in subtree b. The error correcting capabilities of this algorithm are restricted to the identification of substitution transformations in order to preserve the original structure of the tree. A substitution error at node b results from a substitution transformation. This occurs when the node label (terminal symbol) of node b differs from that of the grammar rule (production) applied at node b. For this example, terminal symbols will have only binary values, i.e., they can only have a value of 0 or 1. The total number of substitution errors in subtree b is the sum of the substitution errors in each of node b's children plus the error involved in applying rule k at node b. In the algorithm, subscripts are used to distinguish the substitution errors of each individual child. The range of the subscripts is from one to n, where n is the number of children (rank) of node b. Note that the number of substitution errors accumulates as the tree is traversed from the leaves to the root.

If space is a major consideration, the first element of the triplet, the left-hand side nonterminal, can be eliminated to yield a doublet containing the rule number, k, and the number of substitution errors, e. The extra value used in the triplet, the left-hand side nonterminal, is easily obtained from production rule k, however; this

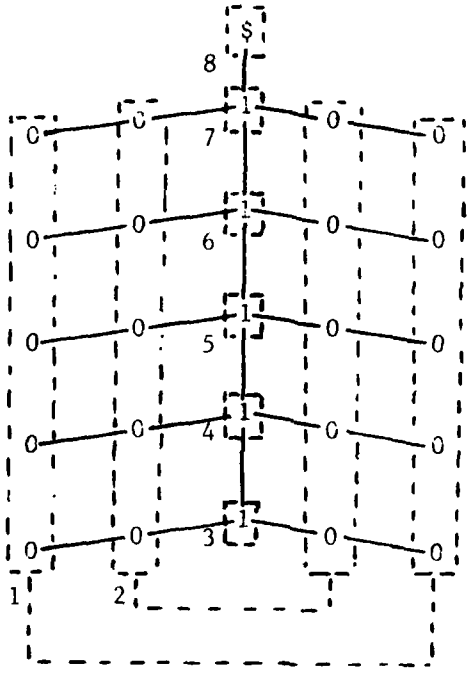


Figure 5.2: Order of processing of leaf node sets for a 5x5 window.

will increase processing time since a search will be necessary to find production rule k and access the left-hand side nonterminal. Triplets are used in this chapter and the following chapters to clarify the processing performed by the algorithm.

The output of the parse will be the triplet tables for every node in the tree. The complete set of triplet tables for the tree is referred to as a tree transition table. The triplets for the final corrected pattern will be chosen from the tree transition table. There will be only one triplet per node in the final result. These triplets are determined by traversing the tree from the root to the leaves starting with the chosen root node triplet and applying its production rule to find the correct triplet for each of its children. This process is continued until the entire tree has been traversed.

An example of the tree transition table for a 3x3 noisy pattern is illustrated in Figure 5.3. The grammar rules used in parsing the tree representation of this noisy pattern are given in Example 3.1. The noisy pattern, its tree representation, and the corrected pattern are diagrammed in Figure 5.4, Figure 5.5, and Figure 5.6, respectively. The pertinent triplets for the corrected pattern are marked by asterisks in the tree transition table.

5.2.2 Processor and Data Allocation

Determining a reasonable processor and data allocation scheme is beyond the scope of this paper. A simulation would be necessary to determine an allocation scheme because of the large number of variables involved. One of the variables involved is the number of independent subtasks within the algorithm. These subtasks depend on the window

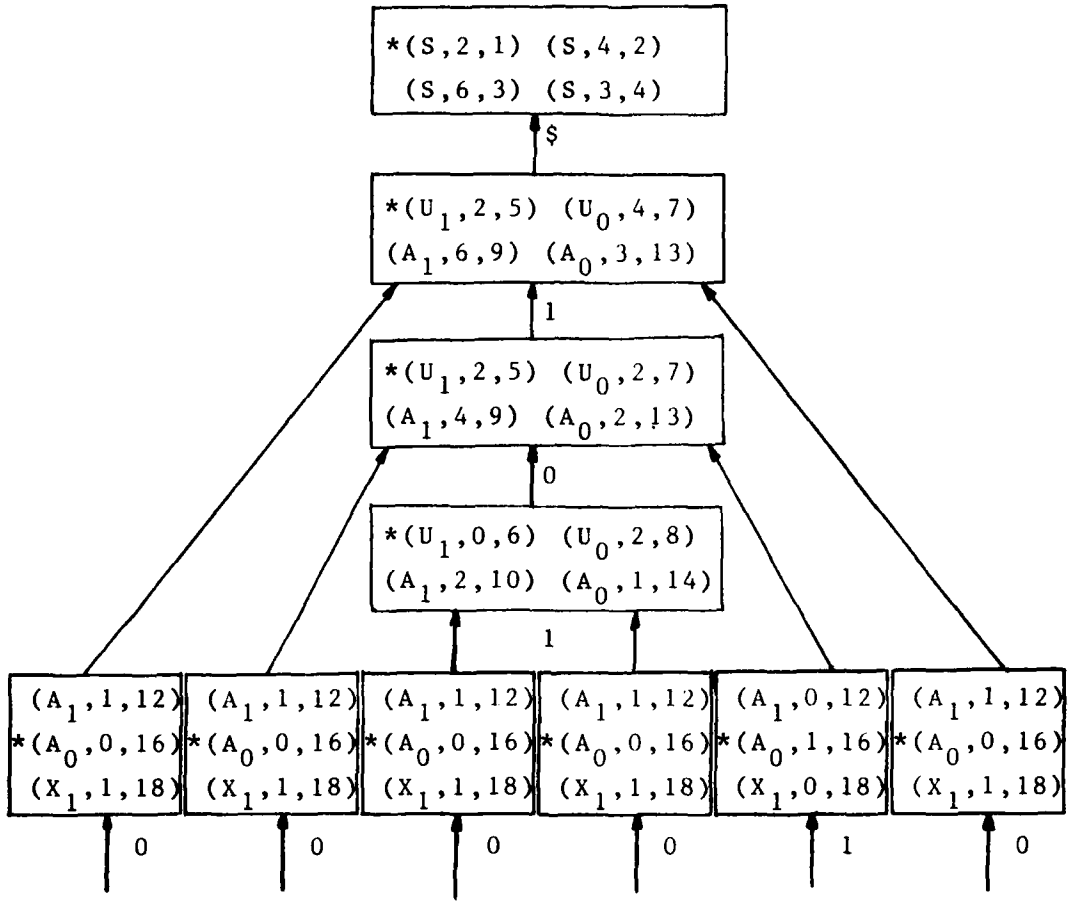


Figure 5.3: Tree transition table for a 3x3 window using the leaf traversal.

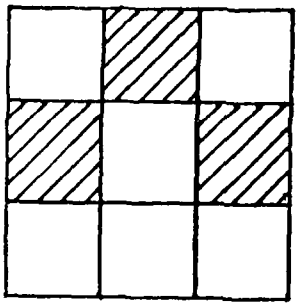


Figure 5.4: A noisy pattern [CF1].

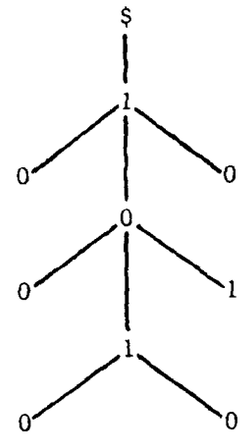


Figure 5.5: Tree representation [CF1].

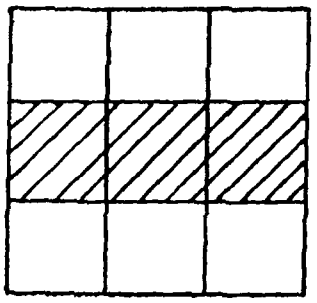


Figure 5.6: Corrected pattern [CF1].

dimensions, the number of grammar rules of each rank, and the number of nonterminal symbols associated with the grammar rules of a particular rank. The number of windows to be processed is another variable to consider. Some reasonable balance needs to be found between the number of windows processed in parallel and the number of processors allocated per window. This assumes that it will not be possible to process all independent subtasks for all windows in parallel.

Considering the problems involved in determining an allocation scheme in greater detail, one might first determine how many independent tasks may be performed in parallel in the different parts of the algorithm. Since for all loops are used to indicate portions of code that can be processed in parallel, Figure 5.1a indicates that all i.f. nodes can be processed in parallel and that for each i.f. node all grammar rules of rank 0 can be processed in parallel. If i is the number of i.f. nodes in a particular WxW window and r_0 is the number of grammar rules of rank 0 for a particular grammar to be used in the parsing process then there are $i \times r_0$ independent subtasks to be performed in the first part of the algorithm in Figure 5.1a.

After processing the grammar rules there will be i tasks to be performed independently. For each i.f. node, redundant states, left-hand side nonterminals, will be deleted. After deleting any redundant states, the last steps of the algorithm in 5.1a involve generating a new set of frontier nodes from the i.f. nodes and obtaining the appropriate subset of the frontier nodes, the lowest level frontier nodes, to be used in the next step of processing. Up to i tasks can be performed independently in these steps.

In the continuation of the algorithm in Figure 5.1b, it should be noted that all l.l.f. nodes can be processed independently. Also, for each l.l.f. node the combinations of nonterminals of its children's triplets can be processed independently and for each combination of nonterminals all grammar rules can be processed independently. If l is the number of l.l.f. nodes in the single WxW window being considered, c_1, c_2, and c_3 are used to indicate the maximum number of triplets that can exist in a child's triplet table and r_1, r_2, and r_3 are the number of grammar rules of rank 1, rank 2, and rank 3, respectively, for the hypothetical grammar used in the first part of the algorithm, then the maximum possible number of independent subtasks:

for n_i (rank of the node) = 1 is $l \times c_1 \times r_1$,

for $n_i = 2$ is $l \times c_1 \times c_2 \times r_2$, and

for $n_i = 3$ is $l \times c_1 \times c_2 \times c_3 \times r_3$.

Referring to Figure 5.2, it should be noted that in the first part of the algorithm, Figure 5.1a, only one serial step is performed, i.e., only one set of 2W nodes in a WxW window is processed. The remaining serial processing steps are performed in the second part of the algorithm, Figure 5.1b. For the nodes of rank 1 (with the exception of the root node) the number of nodes in the set of nodes being processed is 2W. However, for the nodes of rank 2 and rank 3, the trunk nodes, only one node is processed at a time. There is only one element in the set of nodes to be processed. Since the value of l is 1 for nodes of rank 2 and rank 3, the maximum possible number of independent subtasks for $n_i = 2$ could be written as $c_1 \times c_2 \times r_2$, and for $n_i = 3$ the maximum possible number of independent subtasks is $c_1 \times c_2 \times c_3 \times r_3$.

Upon the completion of the processing of the triplet combinations, redundant states will be deleted from the l new triplet tables. At this point there will be l independent subtasks. Following the deletion of the redundant states, the new set of nodes for the next step of processing is determined. This process can involve up to l independent subtasks.

The final step in the algorithm determines if the parse has been successful. This involves a single independent subtask.

The following is a simple example which will illustrate how the number of independent subtasks may vary during the processing of a single window. Using an example from Chang and Fu [CF1,CF2] a 98x98 picture is divided into 196 7x7 windows. These windows are parsed using the grammar rules from grammar G_1. Grammar G_1 has 7 rules of rank 0, 7 rules of rank 1, 10 rules of rank 2, 60 rules of rank 3, and 10 starting rules. It also has 7 nonterminal symbols of rank 1 and 10 nonterminal symbols of rank 2. However, there are only 10 nonterminal symbols of rank 3 while there are 60 rules. But for the nodes of rank 3, triplets with redundant nonterminal symbols are removed during processing leaving the triplet with the least number of errors. Therefore, a maximum of 10 triplets is possible in a triplet table.

The above values will be used in calculating the maximum possible number of children in a triplet table. For the nodes of rank 0, rank 1, and rank 2; the maximum values for the number of triplets are 7, 7, and 10, respectively. The resulting values are as follows:

for $n_i = 0$, $i = 14$, $r_0 = 7$ and $i \times r_0 = 98$, (1)

for $n_i = 1$, $l = 14$, $c_1 = 7$, $r_1 = 7$, and $l \times c_1 \times r_1 = 686$ (2)

or for the root node $n_i = 1$, $l = 1$, $c_1 = 10$, $r_1 = 10$, and $l \times c_1 \times r_1 = 100$ (1)

for $n_i = 2$, $c_1 = 7$, $c_2 = 7$, $r_2 = 10$ and $c_1 \times c_2 \times r_2 = 490$ (1)

for $n_i = 3$, $c_1 = 7$, $c_2 = 10$, $c_3 = 7$, $r_3 = 60$ and $c_1 \times c_2 \times c_3 \times r_3 = 29{,}400$ (6).

The encircled numbers indicate repetition factors within the algorithm. The values obviously vary greatly with the processing of the nodes of rank 3 which by far involve the greater number of independent subtasks. It should also be remembered that these values are for a single window. For the whole picture 196 windows must be processed.

The above discussion is not machine dependent. It involved a consistent method of analyzing the number of independent subtasks in the leaf traversal algorithm. In terms of processor allocation, each independent subtask could be considered to be allocated to a single processor. At this point it is necessary to examine the constraints placed on processor and data allocation by PASM.

The most obvious limiting factor will be the size of PASM, i.e., the number of processors in the machine. For example, if PASM's size is 1024 processors, obviously it is impossible to process even the independent subtasks within a single window in parallel.

For the sake of simplicity, only fixed size allocations of processors would be considered initially in a simulation. The number of processors allocated for a window would be fixed during processing.

For example, a reasonable restriction might be to allow processing of all i.f. nodes and l.l.f. nodes of rank 1 in a node per processor allocation scheme, i.e., 2W processors per window. If less than 2W

processors are allocated then it would seem that additional system overhead would be needed to monitor processor allocation and the incidence of wasted processor resources would be increased. A 7x7 window would have 14 i.f. nodes in the set of i.f. nodes and 14 l.l.f. nodes in each set of l.l.f. nodes to be processed.

The next factor which must be considered is the distribution of data in PASM's processors. Since data transfers are restricted to straight transfers, $+2^i$ mod N and -2^i mod N, the data for related nodes should be readily transferable, i.e., the should be in processors readily accessible through one of these transfers. As a result, some processors will not be used. The number of idle processors for a large machine should, however, not be very great. If the machine size is 64, four 7x7 windows could be processed in parallel. The seven nodes on the left side of each tree for the four windows would be in processors 0-27 while the seven nodes on the right side of the tree would be in processors 32-59. In this way related nodes would be in processors whose numbers differ by 32 or 2^5.

As a result of an allocation scheme like the 2W processors per window scheme, some adjustments will have to be made in the processing. For instance in the processing of the grammar rules for the i.f. nodes and the rank 1 l.l.f. nodes a standard serial searching technique will have to be used for searching the grammar rules depending on the organization of the grammar rule database.

Another problem that will be encountered on almost any machine is allocation of the subtasks for the trunk node processing. As was mentioned previously, only one node of the trunk can be processed at a

time. It would seem to be impractical to restrict the processing of the greatest number of subtasks to only one processor in a strict processor per node allocation. Hence the simplifying assumption was made previously that 2W processors would be allocated for the whole algorithm. In this way at least some of the trunk node subtasks will be performed in parallel. It is further assumed that the subtasks performed in parallel would be those in the triplet combination loop, the second for all loop. The innermost loop which involves the processing of the grammar rules would not be performed in parallel at all. This is consistent with the first part of the algorithm.

One question that may arise is how, if at all, could this processor allocation be determined by a compiler. In the above instance the 2W processors per window seemed reasonable considering the proposed size of the machine and the number of windows to be processed. Upon examining the algorithm, one noticeable problem is that the trunk nodes are processed in the same loop as the rank 1 l.l.f. nodes. The allocation scheme requires a data dependent switch from serial processing of the triplet combination loop for rank 1 l.l.f. nodes to a limited parallel processing of the triplet combinations for the trunk nodes. An additional test statement could be added to the algorithm if the compiler is to generate code which will change the processing of the triplet combinations from serial to parallel when the trunk nodes are being processed. This, however, depends on the role the compiler will play in the processor allocation strategy.

The following discussion uses the term "index set." An "index set" is a collection of values that are to be used as indices for arrays.

Furthermore, the operation on array elements specified by an index set can be done on all elements of the array in parallel (simultaneously). For example, if J is an index set, the loop:

```
     DO 10 I = 1,100
  10 A(I) = 0
```

could be written as:

```
     J = {1:100}
     A(J) = 0.
```

Two dimensional arrays could be handled similarly. For example,

```
     DO 10 I = 1,100
     DO 10 II = 1,50
  10 B(I,II) = 0
```

could be rewritten:

```
       J = {1:100}
      JJ = {1:50}
 B(J,JJ) = 0.
```

Further details are in [MSS].

The fixed allocation strategy as presented here may be interpreted in terms of an index set allocation strategy. For example, there are 2W nodes in the set of i.f. nodes, the first index set. The number of processors allocated would depend on the size of the index set. For this case 2W processors could be allocated. It may, therefore, be possible to eliminate the need for an explicit statement in the program by having the microcontroller that oversees the processing monitor the size of the index sets. When there is only one element in an index set, the microcontroller would determine if there are any additional tasks

that may be performed in parallel by determining if there are additional index sets in inner loops. The microcontroller could then distribute the elements to be processed evenly among the allocated processors if there are more elements in the new index set than processors as is the case for the triplet combinations.

If an index set in an inner loop is much smaller than the number of allocated processors, another possibility might be to search for yet another index set. If another index set is found, i.e., there is another nested for all loop then the cross product of the elements in the set may be formed and the elements of the cross product could be distributed evenly among the processors. However, generating a cross product could be a very inefficient and time consuming process.

One final note concerning the processing of the whole picture will be mentioned here. The proposed temporary solution of using 2W processors in a fixed allocation scheme for the whole algorithm will yield the following results for the processing of the whole picture.

If there are R=r2W processors available (for convenience R is chosen to be a multiple of 2W), then the leaf algorithm should be able to process r windows in parallel. If 1024 processors are available and W=7, then r=73 windows can be processed in parallel. In a 98x98 picture there will be 196 windows, therefore each processor will contain the data for either two or three windows.

5.2.3 Mode of Processing

Once a processor and data allocation scheme have been tentatively identified, the next task is to consider the mode of the parallel processing. The allocation scheme proposed in the previous subsection will be used in this subsection. In the previous subsection, the calculation of the number of windows that can be processed in parallel assumed that PASM had not been partitioned into multiple smaller machines. Whether or not partitioning should occur is another factor which will be considered in this subsection in addition to the SIMD versus MIMD problem.

SIMD processing provides synchronized processing of independent subtasks. The ideal distribution of data among processors operating in SIMD mode would be one where each processor will perform the processing for the same number of independent subtasks, preferably one. SIMD processing also provides synchronized communication between processors through the interconnection network.

MIMD mode allows the processing of different independent subtasks with different instruction streams at the same time. The major problem of MIMD mode processing is the lack of synchronization of the processors and interconnection network. This lack of synchronization, however, also provides greater flexibility and time savings by allowing the overlapping of processing and data transfers through the interconnection network. The flexibility of MIMD mode processing is its greatest asset. MIMD mode processing may be reasonable not only for subtasks with different instruction streams, but also for subtasks with the same instruction stream. If the instruction stream includes conditional

statements that may exclude a fair number of subtasks from the majority of the processing, as occurs with the processing of the triplet combinations for the trunk nodes, then it would seem to be a waste to leave the processors deactivated in SIMD mode for the majority of the processing especially if there is a large number of subtasks to be processed.

The question of partitioning will be considered next before considering the algorithms in any detail. The problem will be to determine how many windows should be processed in a separate machine partition.

For example, PASM could be one machine processing as many windows as possible at the same time. This was assumed at the end of the previous subsection, for simplicity, when determining the number of windows that could be processed in parallel. PASM could also be partitioned so that one window or more would be processed by multiple independent groups of processors.

A possible advantage of partitioning would be closer control of the processing of a window. It seems that a mode switch or changes made in processor allocation during processing might be more efficiently made on a smaller scale. One disadvantage would be the increase in the number of inactive processors. As a result fewer windows could be processed in parallel. For example, if 196 7x7 windows need to be processed, each microcontroller controls 16 processors, the 2W processor allocation scheme discussed in the previous section is used and a one window per microcontroller partition is used then 14 processors would be required to process the i.f. nodes and l.l.f. nodes in a window and there would

be two extra processors in each machine partition. As a result only 64 windows could be processed in parallel if the total number of processors is 1024. This would require each microcontroller to process three and in some cases four windows. Since the machine partition must be a power of two, 16 processors are allocated although only 14 processors are needed. All 16 processors may be used when processing the triplet combinations for the trunk nodes to gain some time savings. This would require additional flexibility in the processor allocation scheme since all 16 processors could not be allocated if a strictly fixed allocation scheme is used.

Now consider the processing of the leaf traversal algorithm in more detail. For this example, the 2W processor allocation scheme will be used and PASM will be considered to be partitioned so that each microcontroller controls an independent machine consisting of 16 processors.

In the above discussion and in the next subsection some assumptions about the organization of the data have been made. It is assumed that an appropriate data structure has been chosen for the grammar rules. It would seem that organization on the basis of rank is of primary importance in decreasing the searching time, while appropriate coding and ordering of the rules within a rank group is of secondary importance. The grammar rules can be organized and stored according to both of these criteria in the lexical analysis phase of processing [CF1,CF2]. In other words, the lexical analysis phase formats the grammar rule database for the picture processing. Note that the lexical analysis phase is performed only once provided that no productions are

added, deleted, or changed in any way after the processing of the picture begins.

It is also assumed that an appropriate data structure is used to store the tree structures for the windows. The data structure should be designed so that it will be easy to point to and access all leaf nodes of an unprocessed tree or a pruned tree whose leaves are lowest level frontier nodes.

During the processing, an entry will be made in the triplet table of every i.f. node for each grammar rule of rank 0. For each grammar rule, every node will determine if a substitution error exists.

A rough outline of how this processing might be performed on PASM is as follows. Each processor will contain sets of node labels for three and in some cases four windows. Each set for each window will consist of one node from each of the groups of nodes as diagrammed in Figure 5.2. The list of grammar rules of rank 0 will be loaded into all of the processor memories. The microcontroller will then broadcast the commands and the processors will be activated and deactivated data conditionally.

In the second part of the algorithm, Figure 5.1b, the processing is similar. Analyzing the processing performed in this loop in further detail, the mode may switch [illegible]ing on the node to be processed. First consider the l.l.f. nodes for which there is only one child, the non-trunk l.l.f. nodes. For these nodes, there are no combinations of triplets to be formed. In addition, the nonterminal symbols in each of the triplet tables for these nodes will be identical. The same amount

of processing will be performed using the same grammar rules for each node.

SIMD mode processing would seem to be the ideal choice for the processing of these i.f. nodes and rank 1 l.l.f. nodes. However, the processing of the triplet combinations of the trunk nodes may not be best performed using SIMD mode.

MIMD mode processing would seem to be a better choice for the processing of the trunk node triplet combinations. MIMD mode should be able to allow the most efficient use of the limited number of processors available. The microcontroller could control the distribution and processing of the triplet combinations so that no processor will be idle until all the triplet combinations have been processed. In processing the trunk nodes, only those grammar rules whose nonterminals match the nonterminals in a triplet combination will produce a new triplet. It is assumed that every processor will contain a copy of all of the grammar rules so that each processor will be able to process its triplet combinations independently. Therefore, in any one group of triplets being processed in parallel, some triplet combinations may be producing new triplets while others are not. The processors of those combinations that are not producing new triplets would have to be inactive for the period of time during which the triplets in the other processors are being formed if SIMD mode processing is used. The processors would be operating on different numbers of different combinations of triplets, hence they would probably operate more efficiently if each processor has the freedom to process its combinations using its own instruction stream in MIMD mode. Thus, for the processing of nodes of rank 0 to 1, SIMD

mode can be used, avoiding the MIMD problems of having synchronization primitives and data contention. In a wxw window, this is w^2-w of the nodes. For the w+1 trunk nodes, PASM can switch to MIMD mode. The 2w processors which could operate in SIMD parallelism for the rank 0 and 1 nodes on a node per processor basis can now be assigned to process the trunk nodes, one node at a time, in MIMD mode.

If MIMD mode is to be used in processing the trunk nodes, and SIMD mode is used in processing the non-trunk l.l.f. nodes, a data dependent mode switch is required. It is assumed that the compiler is capable of generating code that will cause the parallel processing of the triplet combinations for the trunk nodes.

In order for the machine to be able to switch modes, the algorithm may have to explicitly state whether or not a node is a trunk node. The algorithm might need to include a conditional statement distinguishing trunk nodes from those nodes not in the trunk. From this conditional statement, the compiler could probably generate code indicating a mode switch.

A mode switch in the different parts of the algorithm is possible due to PASM's reconfigurability. It is assumed that all of PASM's processors are capable of functioning in both SIMD and MIMD mode.

5.3 The Level Traversal

5.3.1 The Algorithm

This approach was investigated due in part to the results obtained by Chang and Fu [CF1] in a simulation. The simulation performed for their leaf traversal algorithm indicated that a relatively small group

of processors, between four and sixteen depending on the grammar, was the most efficient for processing a window. However, the simulation was not machine specific. If such a small number of processors was to be used it seemed reasonable to try to distribute processing more evenly so other nodes were processed at the same time as the trunk nodes. Therefore, the level traversal algorithm, a modification of Chang and Fu's leaf traversal algorithm was developed.

Another factor to consider is that the results of a simulation for PASM could be very different because of PASM's proposed size and flexibility. If different allocation schemes with greater numbers of processors were efficient on PASM it is possible that a more even distribution of the node processing might allow more windows to be processed in parallel. This might yield significant time savings by reducing the time taken to process the entire picture.

The groups of nodes capable of being processed in parallel and the number of serial steps necessary for processing a 5x5 window using the level traversal are diagrammed in Figure 5.7. The level traversal algorithm is given in Figure 5.8. The tree transition table resulting from the level algorithm for the 3x3 noisy pattern used as an example in section 5.2.1 is given in Figure 5.9. All of the nodes at each level of the tree are processed by the level traversal algorithm in parallel. The operations performed on each node are identical to those of the leaf traversal algorithm. In other words, the processing of individual nodes is identical in both algorithms.

The major difference between the two algorithms is the order of the processing of the nodes. This is evident in Figures 5.3 and 5.9. In

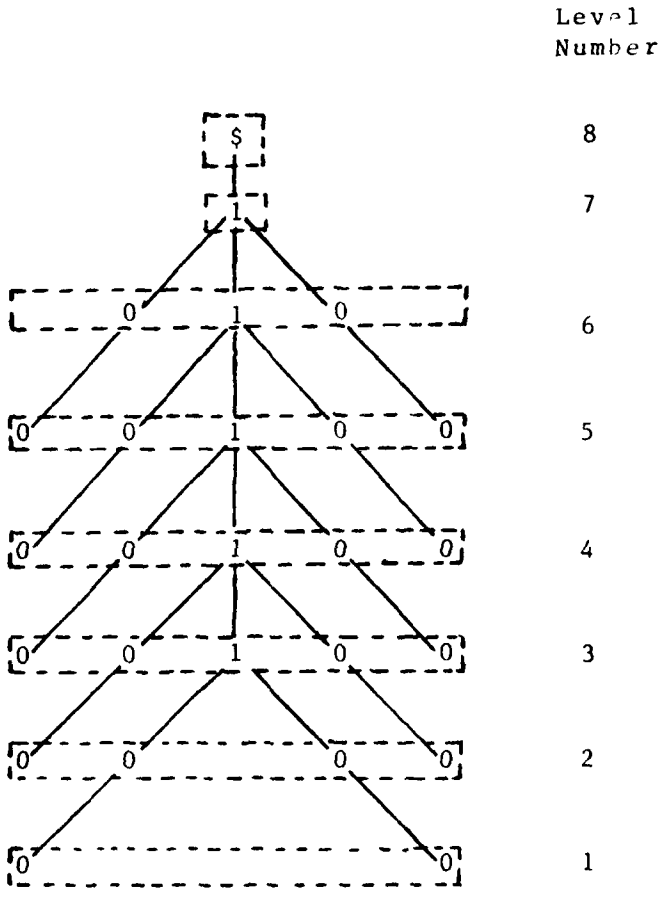


Figure 5.7: Order of processing of level node sets for a 5x5 window.

Algorithm 2. A level traversal algorithm for the implementation of a minimum-distance SPECTA on a parallel machine.

Input: G=(V,r,P,S), the tree grammar, and an input tree B.
Output: Tree transition table for tree B.

```
while the root node has not been processed
begin
   for all node_i where node_i ε {lowest level nodes} of tree B do
   begin
      if node_i is an i.f. node
         then do
         begin
               for all grammar rules with rank = 0 do
               begin

                  if the terminal symbol of rule k = label of i.f. node_i
                        then add triplet (X,0,k) to triplet table of
                              i.f. node_i

                        else add triplet (X,1,k) to triplet table of
                              i.f. node_i

                        /* X is the left-hand side nonterminal of the */
                        /* k^th grammar rule                          */

               end
         end
```

Figure 5.8: The level traversal algorithm.

```
            else do /* the node is a lowest level frontier */
            begin
               n_i = rank of l.l.f. node_i
               for all different combinations of nonterminals of
                        l.l.f. node_i's children's triplets do

               /* Combinations are formed by taking the nonterminal */
               /* of one triplet from each child's triplet table.   */

               begin
                  for all grammar rules with rank = n_i do
                  begin

                     if the nonterminal(s) of rule k = the nonterminals(s)
                        of the triplet combination of l.l.f. node_i
                         then if the terminal symbol of rule k = the label
                                 of l.l.f node_i
                              then add (X,e_1+...+e_n_i,k) to triplet

                                   table of l.l.f. node_i

                              else add (X,e_1+...+e_n_i+1,k) to

                                   triplet table of l.l.f. node_i
                  end
               end
            end

         if the triplet table of any current lowest level node is nonempty
            then if more than one triplet has the same state
               then delete the triplet(s) with the larger number of errors

         lowest level = level of parent nodes of the set of current lowest
                        level nodes
      end
      {current lowest level nodes} = {nodes with level = lowest level}
   end

   if triplet (S,e,k) is in the triplet table of the root node
        then tree B is accepted with e errors
        else tree B is rejected
```

Figure 5.8: (continued)

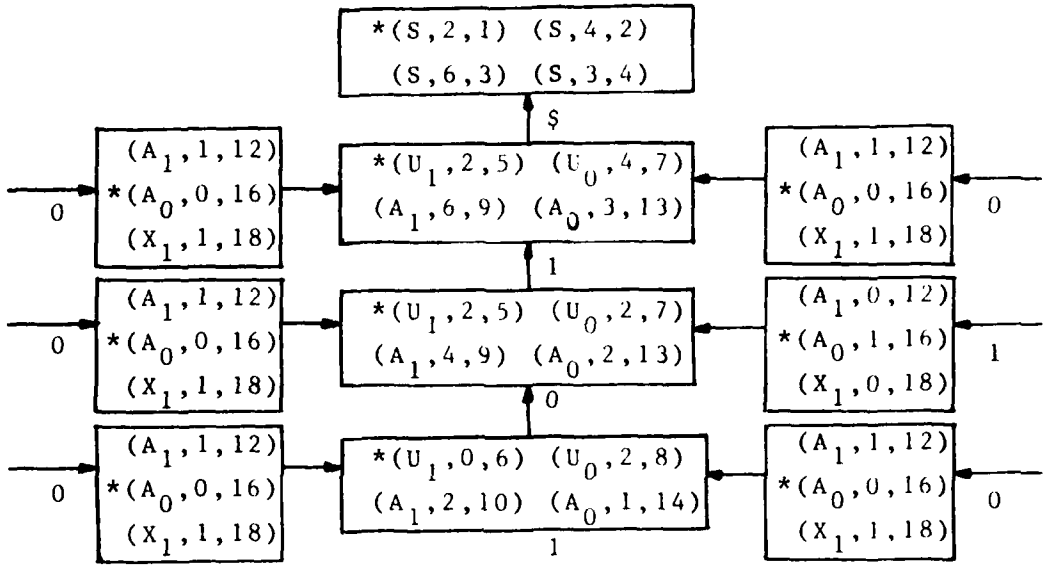


Figure 5.9: Tree transition table for a 3x3 window using the level traversal.

these figures the triplet tables of nodes that may be processed in parallel are aligned horizontally. In the level traversal algorithm, both initial frontier nodes and lowest level frontier nodes are processed in parallel. In contrast, the leaf traversal algorithm processes all initial frontier nodes first, then it processes lowest level frontier nodes.

The notation used in describing the level traversal algorithm is identical to that used in the description of the leaf traversal algorithm, with the exception of the use of the terms lowest level and lowest level nodes. Referring to Figure 5.7, initially the lowest level is level one, the first processing step. Once all the nodes on the current lowest level have been processed the lowest level is updated to the next lowest level (i.e., from level i to level i+1 in Figure 5.7). Any node at the current lowest level is referred to as a lowest level node.

5.3.2 Processor and Data Allocation

Two possible processor allocation schemes for the level algorithm will be considered here. The first is a slight variation of the 2W processor per window allocation scheme suggested in section 5.2.2. The second is a strict processor per node allocation scheme.

The 2W processor per window allocation scheme cannot be used as described in section 5.2.2 because trunk nodes in the level algorithm are processed at the same time as i.f. nodes and rank 1 l.l.f. nodes. The basic principle, however, can be used. In the allocation scheme in section 5.2.2 each node is assigned to one processor. The processing of the i.f. nodes and rank 1 l.l.f. nodes occur with a processor per node

allocation. When a trunk node is processed the triplet combinations are distributed and processed across all of the allocated processors.

For the level algorithm, the i.f. nodes and rank 1 l.l.f. nodes may be processed using a processor per node allocation scheme. If a trunk node is being processed, any extra processors will be allocated to process the trunk node. This is assuming a fixed group of processors has been allocated to the window. The number of processors available for trunk node processing will depend on the size of the group of processors allocated to the window. If 2W processors are allocated then there will be fewer processors available for trunk node processing using the level algorithm than for the leaf algorithm trunk node processing. In the level algorithm W-1 of the 2W processors may be busy processing i.f. nodes and rank 1 l.l.f. nodes, whereas all 2W processors would be available for trunk node processing in the leaf algorithm.

It should be noted, however, that the level algorithm does not need to have 2W processors allocated per window. If the only criterion used in determining the size of the group of processors allocated is to allocate only enough processors so that each node in a set that can be processed in parallel is allocated a single processor then only W processors would be needed for the level algorithm. Although this restricts the amount of parallel processing that can be performed for the trunk nodes, it increases the number of windows that can be processed in parallel. For example, if R=r2W processors are available, then 2r windows may be processed in parallel. If 1024 processors are available and W=7, then at least 2r=146 windows can be processed in parallel.

Consider the trade-off between processing the trunk node triplet combinations in parallel and processing more windows in parallel. If it is best to process as many windows in parallel as possible then a processor per node allocation scheme for all nodes might be used. If the nodes are allocated such that the number of processors available is used to the fullest extent, it may be possible to process all windows in parallel while processing the sets of nodes at certain levels. This will depend on the window size and the number of processors available. For example, if there are 1024 processors and W=7 then only 146 windows of the 196 windows of a 98x98 picture may be processed in parallel for the levels with the largest sets of nodes. The 50 windows left would be processed in a second parallel step. During this time two-thirds of the processors would be idle in a strict processor per node scheme. Thus it would be better to employ a hybrid approach -- processing only half the windows at a time, using the node per processor allocation for the rank 1 and rank 0 nodes and multiple processors allocated for processing the triplet combinations of the trunk nodes.

5.3.3 Mode of Processing

For both of the allocation schemes described in the previous subsection, the processing should be performed in MIMD mode. Nodes being processed in parallel will require different instruction streams and different data streams. For instance, i.f. nodes will be processed at the same time as l.l.f. nodes, and, among the l.l.f. nodes, trunk nodes will be processed at the same time as nodes not in the trunk. The code for processing the different types of nodes is different, and the different types of nodes require the accessing of different parts of the

grammar rule database. Assuming all the processors in PASM are to be used for the processing of a picture, they will all be capable of operating in MIMD mode.

The grammar rule database should be organized as was suggested for the leaf algorithm. The tree data structure for the window, however, will need to be different to allow easy access to lowest level nodes on a complete tree or on a pruned version of a complete tree.

5.4 A Comparison

For the sake of simplicity, a processor per node allocation scheme for both algorithms will be considered first. Unless processing in one mode is found to be significantly more efficient than processing in the other mode, there should be only a slight difference in the time taken by the algorithms in the processing of an individual window. If the processing in both modes is equally efficient, then the time for processing a single window using a processor per node allocation scheme should be the same for both algorithms.

Using as an example, the 5x5 window as illustrated in Figure 5.7, for the level traversal tree, at the lowest level all the nodes are i.f. nodes, as are all the leaves in the leaf traversal tree. The time to process any one i.f. node is the same for all i.f. nodes, therefore the first step will take the same amount of time. Processing of l.l.f. nodes will take longer than the processing of i.f. nodes because it is necessary to compare the nonterminal symbols of its children's triplets with the nonterminal symbols of the grammar rule. Therefore, the second step of the level algorithm will take as long as is needed to process a l.l.f. node. All the non-trunk l.l.f. nodes take the same amount of

time to process as was noted in the analysis of the leaf algorithm. The second step in the leaf algorithm will, therefore, take the same amount of time as the second step in the level algorithm. This same principle applies for the remainder of the processing steps. Although it is demonstrated in Figure 5.2 and 5.7 for 5x5 window, this principle is generally applicable across all window sizes. The mode of processing should be the only factor that would make a difference in the processing time of the two algorithms for a single window.

There is a large difference, however, in the number of windows that can be parsed in parallel given a processor per node allocation scheme. Given R=2rW processors, the level algorithm can process 2r windows in parallel, while the leaf algorithm can process only r windows in parallel. It is possible, therefore, for the level algorithm to process a whole picture as much as twice as fast as the leaf algorithm. The exact time savings will depend on the relationship of the number of nodes in all windows to be processed in parallel to the number of processors available, assuming a processor per node allocation scheme.

If the trunk nodes can be processed using more than one processor, there may be a difference in the processing time for these nodes. In the leaf algorithm, one trunk node is processed at a time while in the level algorithm other nodes are processed with the trunk nodes. Therefore, for the level algorithm, fewer processors will be available for the processing of the trunk nodes. However, the level algorithm has the potential to process more windows in parallel than the leaf algorithm. An analysis would have to be performed to determine whether

or not these factors would create a significant difference in the processing times for the two algorithms.

CHAPTER 6

DATABASES AND TREE PARSING

6.1 Introduction

Databases and database languages have been designed for the organization of the reasonably complex relationships among data and for the nonarithmetic processing of the data. The two algorithms discussed in the last chapter, the leaf algorithm and the level algorithm, are characteristic of many syntactic pattern recognition tasks in that they use logical data structures more complex than the arrays usually used in image processing tasks and they use many nonarithmetic operations while processing the data in these logical data structures. Therefore, it seems reasonable to use database techniques for the processing of these algorithms. The problem to be considered in this chapter will be the choice of a database model and the feasibility of programming using database techniques on PASM.

Of the three database models, only the network and the relational model were considered in detail. The hierarchical model was not used since it is essentially a subset of the network model. Of the two models presented in detail, only the relational model was investigated fully at the programming level. It was thought that whereas the network model allowed the structuring of the data in the logical tree structure

of the algorithms, it was not as amenable to parallel processing on PASM as the relational model.

An attempt was made to write programs for the algorithms to determine the types of commands that would be needed and how PASM might execute these commands. Although a parallel language for PASM has not been developed, some instruction and declaration statements for SIMD mode processing have been suggested [SiM,MSS]. These are listed in Figure 6.1.

Commands from a database language, GPLAN's DML for the network model and SEQUEL for the relational model, were mixed with the commands in Figure 6.1. The result is a pseudo language with certain inconsistencies, but these will be noted. It is assumed that the pseudo language may be used for both SIMD and MIMD processing. The programs written in this pseudo language serve the purpose of indicating where special language structures may be needed and how these may be processed.

6.2 The Network Approach

The network approach was not fully investigated. The tentative network model developed is illustrated in Figure 6.2. It should be noted that at this stage in the development of the model the grammar rules had not been included in the model of the tree structure. They were established as a second database. In a final network model it would probably be necessary to have the grammar rules included in the tree database. Subsequent references to the model will be concerned only with the model of the tree structure as presented in Figure 6.2.

A. Declaration Statements

```
CONSTANT v1=con1, v2=con2, ..., vN=conN;
INTEGER varlist;
BYTE varlist;
UNSIGNED BYTE varlist;
INDEX varlist;
DATA INPUT varlist1 OUTPUT varlist2;
ivar= {rangelist};
```

B. Control Statements

Micro Controller

```
WHILE condition DO statement
FOR initialize statement WHILE condition
     STEP increment statement DO statement
IF condition THEN statement
IF condition THEN statement1 ELSE statement2
```

PCU Processor

```
WHERE condition DO statement
WHERE condition DO statement1 ELSEWHERE statement2
```

Micro Controller and PCU Processor

```
WHILE condition1 WHERE condition2 DO statement
FOR initialize statement WHILE condition1 WHERE condition2
     STEP increment statement DO statement
WHILE[ANY,ALL] condition DO statement
FOR initialize statement WHILE[ANY,ALL] condition
     STEP increment statement DO statement
IF[ANY,ALL] condition THEN statement
IF[ANY,ALL] condition THEn statement1 ELSE statement2
```

Figure 6.1: List of control instructions [SiM].

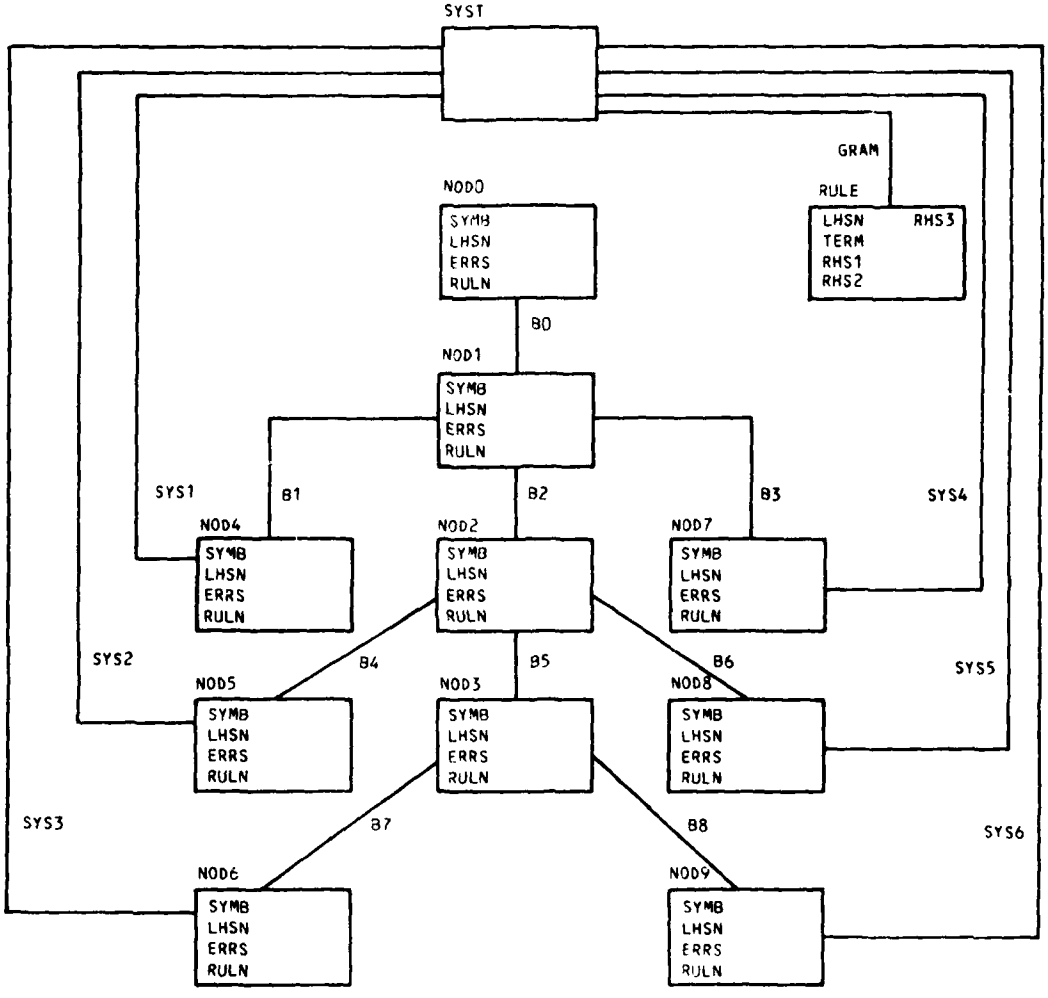


Figure 6.2: Network model of a tree database for a 3x3 window.

The model illustrated in Figure 6.2 includes a unique predefined SYST record as required by GPLAN. Containing no data, the SYST record provides the entry point to the user defined database.

A small amount of redundancy was built into the node records for the purpose of increasing accessing efficiency and keeping the data model as simple as possible. For each occurrence of a node record there will be a node label, SYMB, and a triplet, LHSN, ERRS and RULN. The node lable will be repeated as many times as there are triplets. The node label is such a small data item the redundancy does not greatly increase the storage space required. In addition, since the node label is always used when forming a new triplet, it seemed reasonable to place these logically related data items in the same record.

There are multiple entry points to the database as indicated by the branches SYSn. The entry points are at the bottom of the tree because the algorithms describe a bottom-up process. The multiple entry points are essential for parallel processing. The entry points may all be processed at one time or subsets of entry points may be chosen. The multiple entry points while allowing parallel entry points also allow the user some flexibility in choosing the order in which the nodes are processed. As a consequence, this data model provides all the relationships necessary for both algorithms.

The schema and subschemas for the algorithms are derived from this model. Haseman and Whinston [HaW] give the following definitions of schema and subschema as used with respect to a CODASYL database.

> schema - "The description of the logical structure of a database in terms of a data description language."

subschema - "A particular programmer's view of the logical structure of a database. One schema may have numerous subschemas associated with it."

Although the CODASYL standard includes subschemas, the GPLAN DDL (Data Definition Language) does not include subschemas. The logical restructuring of the tree database that occurs when switching from the leaf algorithm to the level algorithm makes the inclusion of the commands for writing subschemas necessary. Therefore, the CODASYL standard commands for subschemas taken from the SEED [Ger] database management system were included as an extension of GPLAN's DDL.

The terms used in the schema are RECORD, ITEM, SET, OWNER, and MEMBER. ITEM is used to indicate a data item name. RECORD is used to indicate a record name. A SET is named relationship of record types where one or more record types are defined as OWNERs and one or more record types are defined as MEMBERs [HaW]. In the logical structure, the SETS are depicted by the branches connecting the record types.

Figure 6.3 is an example of a very simple schema and its corresponding network model without the GPLAN SYST record. This example should help clarify the relationship between a schema and a diagram of the logical structure of a database.

The schema for the tree database whose logical structure is diagrammed in Figure 6.2 is given in Figure 6.4. The schema does not, by itself, clearly indicate which nodes may be processed in parallel. Some assumptions might be made based on the multiple entry points, but these would be true only for specific algorithms. Multiple entry points by themselves do not imply that all the associated records may be processed in parallel.

```
SCHEMA=EX

RECORD    SENT
ITEM      WRDS

RECORD    WORD
ITEM      ALPH

SET S1
OWNER SENT
MEMBER WORD
```

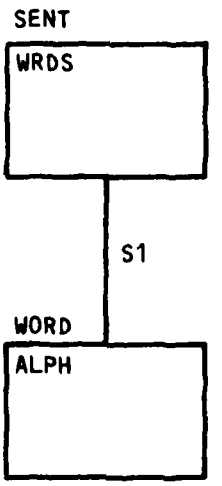


Figure 6.3: A simple schema and corresponding diagram.

```
SCHEMA=TREE

RECORD    NOD1
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD2
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD3
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD4
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD5
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD6
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD7
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD8
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD9
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    NOD0
ITEM      SYMB
ITEM      LHSN
ITEM      ERRS
ITEM      RULN

RECORD    RULE
ITEM      LHSN
ITEM      TERM
ITEM      RHS1
ITEM      RHS2
ITEM      RHS3

SET       B0
OWNER     NOD0
MEMBER    NOD1

SET       B1
OWNER     NOD1
MEMBER    NOD4

SET       B2
OWNER     NOD1
MEMBER    NOD2

SET       B3
OWNER     NOD1
MEMBER    NOD7

SET       B7
OWNER     NOD2
MEMBER    NOD5

SET       B5
OWNER     NOD2
MEMBER    NOD3

SET       B6
OWNER     NOD6
MEMBER    NOD8

SET       B7
OWNER     NOD3
MEMBER    NOD6

SET       B8
OWNER     NOD3
MEMBER    NOD9

SET       GRAM
OWNER     SYST
MEMBER    RULE

SET       SYS1
OWNER     SYST
MEMBER    NOD4

SET       SYS2
OWNER     SYST
MEMBER    NOD5

SET       SYS3
OWNER     SYST
MEMBER    NOD6

SET       SYS4
OWNER     SYST
MEMBER    NOD7

SET       SYS5
OWNER     SYST
MEMBER    NOD8

SET       SYS6
OWNER     SYST
MEMBER    NOD9
```

Figure .4: Schema for the tree database.

The easiest method of letting the compiler know which sets and, therefore, which set record MEMBERS may be processed in parallel is to have the programmer tell the compiler in the program. This does not entail any knowledge of the hardware on the part of the programmer and it does not require any specific knowledge about the compiler and processor allocation. What is proposed is a method whereby the programmer simply states, based on the algorithm, which sets can be processed in parallel using a new database command for denoting parallel set grouping.

The keyword for defining a parallel group name is PARALLEL. The sets in a parallel group are listed following the group name definition.

Theoretically, for database security, the schema is a fixed entity that the programmer is not allowed to change. Therefore, if the programmer is to be able to define the parallel groups that are needed, this must be accomplished in the subschema. Placing the parallel set definitions in the subschema also allows more flexibility in the use of a database. Since the schema is fixed, the only opportunity for flexibility, for different interpretations of different problems on the part of the user, is in the subschema.

The subschemas for the two algorithms are given in Figure 6.5. The first line of the subschemas give the subschema name and the name of the schema. The next two lines of each subschema indicate which records and which sets of the schema will be used. In this case all of the records and all of the sets of schema TREE will be used in both of the programs. The parallel groups are listed following the record and set listings. All possible parallel paths are listed in the parallel groups. It

```
SUBSCHEMA=LEAF, SCHEMA=TREE
COPY ALL RECORDS
COPY ALL SETS
PARALLEL GPL1
   SYS1
   SYS2
   SYS3
   SYS4
   SYS5
   SYS6
PARALLEL GPL2
   B1
   B3
   B4
   B6
   B7
   B8

SUBSCHEMA=LEVEL, SCHEMA=TREE
COPY ALL RECORDS
COPY ALL SETS

PARALLEL GRP1
   SYS1
   B5
   SYS4
PARALLEL GRP2
   SYS2
   B7
   B8
   SYS5
PARALLEL GRP3
   SYS3
   SYS6
PARALLEL GRP4
   B1
   B2
   B3
PARALLEL GRP5
   B4
   B5
   B6
PARALLEL GRP6
   B7
   B8
```

Figure 6.5: Subschemas for the leaf and level algorithms.

should be noted that a set is a part of a parallel group if either an owner record is a record that may be processed in parallel with other records or a member record may be processed in parallel with other records. This is done so that parent nodes can be made frontier nodes in parallel.

The GPLAN DML commands to be used in the programs are not oriented to parallel processing. The inclusion of the parallel groups in a subschema raises some questions as to whether modifications to the GPLAN commands may be needed. Whether or not some modifications are made depends on determining whether the compiler or the user is responsible for finding the parallelism in the program.

If the programmer is responsible for finding the parallelism, parallel commands will be needed. Commands involving a parallel group would be prefaced by a P and the group name would be used instead of a set name. For example, the command FFM (Find First Member) is used to find the first member of a set. Eliminating the hollerith formatting from the standard GPLAN command, the command might be found in a program as follows:

```
CALL FFM S1
```

If S1 is a member of a parallel group, GRP1, the programmer would modify the command and use a Parallel Find First Member (PFFM) command that uses the group name as the argument. The PFFM command would find the first member of every set in the group. For set S1 in GRP1 the resulting command is

```
CALL PFFM GRP1
```

If the compiler is responsible for detecting the parallelism in a program, it will have to use the parallel group information in the subschema. The compiler will check for the parallel group sets. Once a set in a parallel group is found the compiler will check to see if it is within a loop performing the same operations on a different record in other sets in the parallel group. Assuming the programmer letting the compiler detect parallelism is not concerned with parallelism, it is necessary to check for all the sets in a parallel group because the programmer may use a parallel group in one section of a program and an individual set that is a member of a parallel group in another section. However, it will be assumed that the programmer will be able to use the parallel commands, therefore making it unnecessary to include the capacity for detecting whether or not a set is a member of a parallel group in the compiler for PASM.

At this point in the analysis it seemed that the relational approach may be better suited to the task in question. Some of the factors affecting the decision to investigate the relational approach were the clarity of the program and the degree to which parallelism may be exploited.

When speaking of the clarity of a program, the reference is to the clarity of a program written in a host language with database language commands embedded in the program. It is assumed that the relational database language used will be an English-like language similar to SEQUEL. A CODASYL DML, such as GPLAN's DML, that is designed to be embedded in a host language usually consists of subroutine calls using

acronyms for a command function as the command subroutine name and a parameter list. In addition, for each step in the tree processing more CODASYL DML commands are required than relational language commands. The result of using an English-like relaticnal language whose commands can be directly embedded in a host language program should be greater clarity and ease of programming.

The primary reason for switching to the relational approach was the ability of the relational database to exploit parallelism. In a CODASYL database, access strategy is forced to be sequential because of the method of accessing data one occurrence at a time through a chain of pointers. Even with the parallelism allowing the user to traverse multiple chains at the same time, a relational database has the potential ability to access whole sets of record or data occurrences at the same time. The investigation of the relational approach follows in the next section.

6.3 The Relational Approach

6.3.1 Justification of the Relational Approach

The relational approach has several advantages over the network approach. Two of these mentioned previously were the clarity of the relational data languages, in contrast to the CODASYL DML, and the ability to exploit parallelism. Both of these are either direct or indirect results of the uniformity of the data representation which is the outstanding characteristic of the relational data model. The ability to exploit parallelism is a direct result whereas the clarity of the data languages is an indirect result.

The clarity of the relational data languages is due in part to their nonprocedural characteristics and the few simple but powerful set operators which are used to manipulate the data in the relational database. It is not necessary to specify how to obtain data when using a relational database language as it is when using a CODASYL or hierarchical DML. When using a CODASYL or hierarchical DML not only the path but the currency of the path, the current record occurrences on the access path, to a particular data item or record must be specified by the user. A relational data language does not need commands to specify a path because there is no path to follow due to the uniform representation of the data. The user only has to specify the characteristics of the data desired, and if necessary, use the few set operators to obtain the required data.

The relational approach, in general, is far simpler than the network approach. This is particularly true when working in a parallel machine environment. Not only does the relational approach offer greater opportunities for exploiting the parallel processing capabilities of such a machine, but the added simplicity for the user seems to be a powerful factor in support of the use of the relational approach.

6.3.2 The Tree Database Relations

The relations used for the implementation of both algorithms are diagrammed in Figure 6.6. Some of the factors used in determining the relations were the elimination of redundancy within the relations, the nature of the data in the relation, and the logical relationships of the attributes. These factors influenced the organization of the relations

TREE				
NODE#	SYMB	PARENTNODE#	LEVEL#	CURRENT

CHILDREN	
NODE#	CHILDNODE#

GRAM				
RANK	RHSN	TERM	LHSN	RUL#

TRIPLETS			
NODE#	LHSN	ERR	RUL#

COMB	
RHSN	ERR

TEMP			
PN#	CN#	LHSN	ERR

Figure 6.6: The tree database relations.

more than the strict normalization process usually advocated. A relation is said to be normalized if each attribute in each tuple is atomic (nondecomposable) [Dat]. There are, however, multiple levels of normalization. A relation satisfying the above condition for normalization is said to be in first normal form. Currently the most desired form of a relation, the form in which undesirable properties affecting insertion, deletion, and updating have been eliminated, is fourth normal form [Dat]. Some of the relations are in fourth normal form, but only as a consequence of aligning nonredundant, logically related data in a single relation.

The TREE relation contains the node numbers, NODE#, the terminal symbol associated with the node number, SYMB, the parent node number, PARENTNODE#, the level of the node, LEVEL#, and a currency flag, CURRENT. CURRENT has a value of 1 if a tuple is being processed, otherwise the value is 0. The CHILDREN relation lists the nodes, NODE#, and their children, CHILDNODE#. The GRAM relation contains the grammar rules, the rank of the grammar rules, RANK, and the rule numbers, RUL#. The grammar rules consist of a right-hand side nonterminal, RHSN, a terminal symbol, TERM, and a left-hand side nonterminal, LHSN. The TRIPLETS relation contains the node numbers, NODE#, and the triplets associated with the nodes. The triplets consist of left-hand side nonterminals, LHSN, an error count, ERR, and the grammar rule number applied to form the triplet, RUL#. The COMB relation is formed from the TEMP relation. The TEMP relation contains the parent node numbers, PN#, the child node numbers, CN#, the left-hand side nonterminals obtained from the TRIPLETS relation, LHSN, and the error counts, ERR, associated

with a node's children's triplets. The COMB relation contains the new right-hand side nonterminals formed by combining the LHSNs in TEMP, RHSN, and error counts, ERR, formed by adding the appropriate ERRS values in TEMP. It should be noted that RHSN is a variable size attribute which can contain from zero to three nonterminal symbols.

The relations can be divided into two groups: static and dynamic. A static relation is one in which attribute values are not updated, inserted, or deleted. A dynamic relation is one whose attribute values are at some time either inserted, deleted, or updated. Usually in a database the relations are dynamic, but for the tree algorithms the tree structure and grammar rules are considered to be set, static, once the database is loaded. New tree structures or grammar rules can be used without reloading the entire database, but it would be necessary to reload the relations affected.

The static relations are TREE, CHILDREN, and GRAM; the dynamic relations are TRIPLETS, COMB, and TEMP. Of the static relations, TREE and CHILDREN are in fourth normal form. For GRAM to be in fourth normal form it would have to be divided into two relations; one relation would contain the attributes RUL#, LHSN, TERM, and RHSN, while the other would contain the attributes RUL# and RANK. GRAM was not put in fourth normal form because it was much easier to have the logically related attributes joined in a single relation. Whether or not GRAM is in fourth normal form is not crucial because it is a static relation; therefore, the undesirable properties eliminated by placing a relation in fourth normal form do not occur.

Of the dynamic relations, COMB and TEMP are in fourth normal form. TRIPLETS will be in fourth normal form if the attribute LHSN is removed, but because the SEQUEL commands used do not perform joins easily the LHSN attribute was left in the TRIPLETS relation. The dynamic relations as constructed are not necessarily the most efficient, but they are sufficient to demonstrate the usage of a relational database for image processing. Obviously using relations will require extra knowledge on the part of the user. The user should know something about the normalization process and have some concept of the limitations of the relational language used.

6.3.3 Relation Declarations

Once the logical structure of the relations is specified the user will need to write the relational schema. Instead of having a strict schema with its own DDL, a much simpler approach would be to add a new data type that can be declared along with the regular PASM declarations. The data type would be called RELATION. The declaration of the CHILDREN relation would appear as follows:

RELATION CHILDREN [W^2] {INTEGER NODE#, CHILDNODE#}

The length of the relation, CHILDREN, is W^2, where W is the window dimension. This value is included because some value must be given for the maximum length of the relation even if the relation is dynamic.

This declaration uses the proposed PASM INTEGER declaration. Any of the other PASM data types may conceivably be used within a RELATION declaration. The proposed PASM data declarations do not include a character data declaration. However, syntactic pattern recognition

tasks frequently use character data. Encoding of the character data is often performed, but theoretically, to allow the user the greatest freedom, there should be a CHAR, character, data declaration.

The most important restriction on a RELATION declaration is that the user cannot nest RELATION declarations. A relation must contain all atomic values. Multiple levels of declarations result in composite attributes and, therefore, are not allowed.

By not having a formal schema, a certain amount of flexibility is lost because the user can no longer specify which attributes are to be keys. In the RELATION declaration, all attributes are candidate keys; there is no specific primary key. This does, however, make it slightly easier for the user.

6.3.4 A Brief Comparison with the CODASYL Approach

At this point, a brief comparison with the CODASYL approach will illustrate many advantages of the relational approach and a few disadvantages. One primary advantage of the relational approach, the ability of the relational languages to extract whole sets of data, was mentioned previously. Another advantage of the relational approach is the simplicity of the relational "schema." Use of the CODASYL schemas and subschemas would require much more knowledge on the part of the user and also would require more processing. The CODASYL schemas and subschemas are far more complex than the relation declarations. This is primarily due to the greater complexity of the logical structure for the CODASYL approach. The logical relational structure is much simpler even with the extra relations, COMB and TEMP, added to help form the triplet combinations.

The one major disadvantage of the relational approach lies in the initial accessing strategy. Whereas the CODASYL entry points allow a flexible approach, i.e., the user can choose the set of entry points, the relational approach required on additional attribute to process the nodes in a different order. The PARENTNODE# can be used to process the nodes in the leaf algorithm, but to process the nodes by level the LEVEL# attribute had to be added. This inflexibility is minor when compared with the previously mentioned disadvantages of the CODASYL approach.

6.3.5 The Leaf and Level Programs

To simplify the transition from algorithms to tentative programs, flowcharts are used. The flowchart for the leaf algorithm is given in Figure 6.7. The corresponding program is given in Figure 6.8. The flowchart and program for the level algorithm are given in Figures 6.9 and 6.10, respectively. The relational language used in the programs is based upon SEQUEL with a few minor modifications added to help make the programs more readable. Among these changes were the use of full relation references at all times. For example, SELECT NODE# FROM TREE, a perfectly acceptable reference in SEQUEL, will be written SELECT TREE.NODE# FROM TREE. The host language used in the program is the proposed parallel language for PASM. The commands for this proposed language were listed previously in Figure 6.1. The function of some SEQUEL commands used in the leaf and level programs will be described next.

Retrieval operations are usually performed in SEQUEL with a SELECT-FROM-WHERE* block. This command SELECTs attributes FROM a

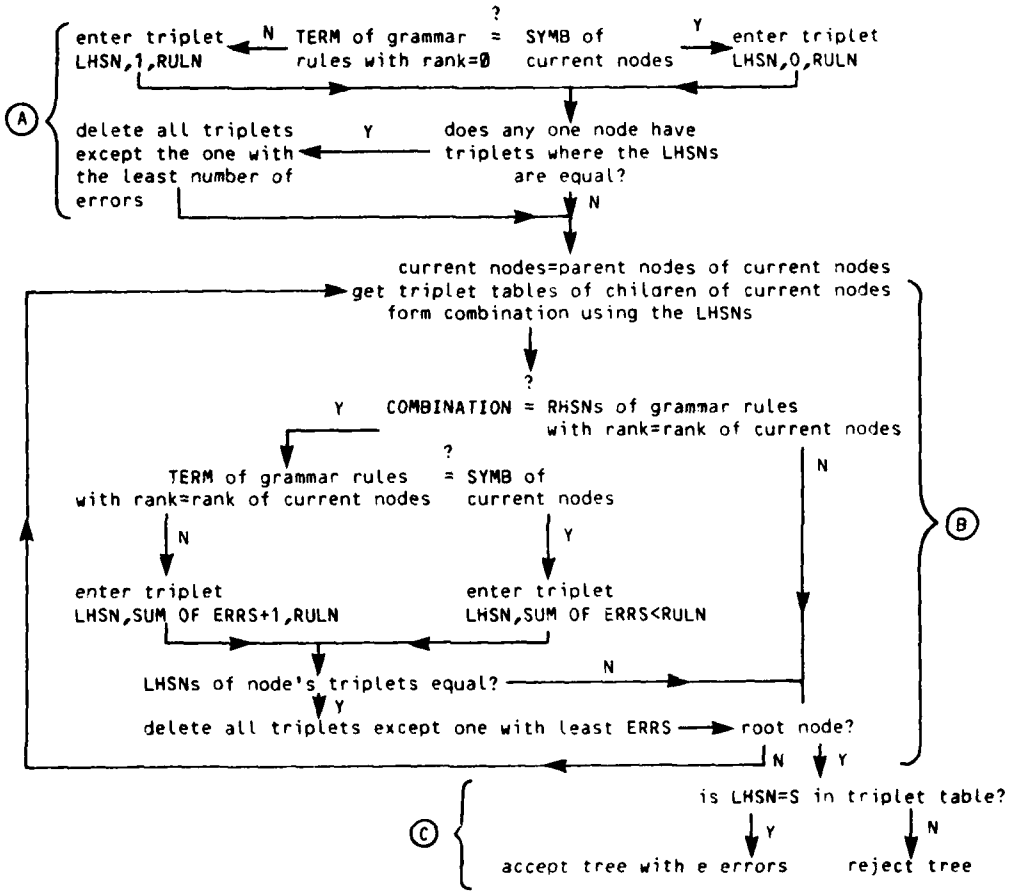


Figure 6.7: Flowchart for the leaf algorithm.

```
A  WHERE  SELECT GRAM.TERM    =   SELECT TREE.SYMB
          FROM GRAM               FROM TREE
          WHERE* GRAM.RANK=0      WHERE* TREE.CURRENT=1

     DO   INSERT INTO TRIPLET:
            (TREE.NODE#,GRAM.LHSN,1,GRAM.RUL#)

    ELSEWHERE
          INSERT INTO TRIPLET:
            (TREE.NODE#,GRAM.LHSN,0,GRAM.RUL#)

  DELETE TRIPLETS
  GROUP BY TRIPLETS.NODE#
  GROUP BY TRIPLETS.LHSN
  HAVING TRIPLETS.ERR    =
          SELECT MIN(TRIPLETS.ERR)
          FROM TRIPLETS

WHERE TREE.CURRENT=1 AND TREE.NODE#  =0
  DO  UPDATE TREE
      SET TREE.CURRENT=1
      WHERE* TREE.NODE# IN
         SELECT TREE.PARENTNODE#
         FROM TREE
         WHERE* TREE.CURRENT=1
         UPDATE TREE
         SET TREE.CURRENT=0
         WHERE* (SELECT TREE.NODE#
                 FROM TREE
                 WHERE* TREE.CURRENT=1)
                 INTERSECT
                (SELECT TREE.PARENTNODE#
                 FROM TREE
                 WHERE* TREE.CURRENT=1)
```

Figure 6.8: The leaf algorithm program.

```
B  WHERE SELECT COUNT (CHILDREN.CHILDNODE#)=1
         FROM CHILDREN
         GROUP BY CHILDREN.NODE#
         HAVING SET (CHILDREN.NODE#)
         CONTAINS (SELECT TREE.NODE#
                   FROM TREE
                   WHERE TREE.CURRENT=1)
     DO  INSERT INTO COMB:
            SELECT TRIPLETS.NODE#,TRIPLETS.LHSN,TRIPLETS.ERR
            FROM TRIPLETS
            WHERE* TRIPLETS.NODE#=CHILDREN.CHILDNODE#

   ELSEWHERE  DO
      TEMP (PN#,CN#,LHSN,ERR) ← SELECT UNIQUE
                                CHILDREN.NODE#,CHILDREN,CHILDNODE#,
                                TRIPLETS.LHSN,TRIPLETS.ERR
                                FROM CHILDREN,TRIPLETS
                                GROUP BY CHILDREN.NODE#
                                GROUP BY CHILDREN.CHILDNODE#
                                HAVING COUNT (CHILDREN.CHILDNODE#)
                                       > 1 AND
                           SET (CHILDREN,CHILDNODE#) CONTAINS
                             SELECT CHILDREN.CHILDNODE#
                             FROM CHILDREN
                             WHERE* CHILDREN.CHILDNODE# =
                                TRIPLETS.NODE# AND
                                CHILDREN.NODE# IN
                                  SELECT TREE.NODE#
                                  FROM TREE
                                  WHERE* TREE.CURRENT=1
      RANK ← SELECT COUNT (UNIQUE TEMP.CN#)
             FROM TEMP

      T(N) ← SELECT COUNT (TEMP.LHSN)
             FROM TEMP
             GROUP BY CN#
```

Figure 6.8: (continued)

```
                FOR I=1 WHILE I <= T(1) STEP I=I+1  DO
                 FOR J=T(1) WHILE J <= T(1)+T(2) STEP J=J+1  DO
                  IF RANK=3  THEN
                    FOR K=T(1)+T(2) WHILE K <= T(1)+T(2)+T(3)
                                    STEP K=K+1  DO
                      RHSN = LHSN(I)||LHSN(J)||LHSN(K)
                      ERR = ERR(I)+ERR(J)+ERR(K)
                      INSERT INTO COMB:
                        (RHSN,ERR)
                  ELSE
                     RHSN = LHSN(I)||LHSN(J)
                     ERR = ERR(I)+ERR(J)
                     INSERT INTO COMB
                       (RHSN,ERR)
                DELETE TEMP

            WHERE  SELECT GRAM.RHSN
                   FROM GRAM
                   WHERE* GRAM.RANK IN
                     SELECT COUNT (CHILDREN.CHILDNODE#)
                     FROM CHILDREN
                     GROUP BY CHILDREN.NODE#
                     WHERE* CHILDREN.CHILDNODE# IN
                      SELECT TREE.NODE#
                      FROM TREE
                      WHERE* TREE.CURRENT=1
                 DO WHERE GRAM.TERM = SELECT TREE.SYMB
                                      FROM TREE
                                      WHERE* TREE.NODE# = COMB.NODE#
                      DO INSERT INTO TRIPLET
                           (TREE.NODE#,GRAM.LHSN,COMB.ERR+1,GRAM.RUL#)

                   ELSEWHERE  INSERT INTO TRIPLET
                           (TREE.NODE#,GRAM.LHSN,COMB.ERR.GRAM.RUL#)

                   DELETE TRIPLETS
                   GROUP BY TRIPLETS.NODE#
                   GROUP BY TRIPLETS.LHSN
                   HAVING TRIPLETS.ERR   =
                           SELECT MIN(TRIPLETS.ERR)
                           FROM TRIPLETS
                   DELETE COMB
```

B cont.

Figure 6.8: (continued)

```
C   ELSEWHERE DO
      IF ANY TRIPLETS.NODE#=0 AND TRIPLETS.LHSSN='S'
         THEN INSERT INTO RESULT:
                ('ACCEPT',TRIPLETS.ERR)
         ELSE INSERT INTO RESULT:
                ('REJECT')
```

Figure 6.8: *(continued)*

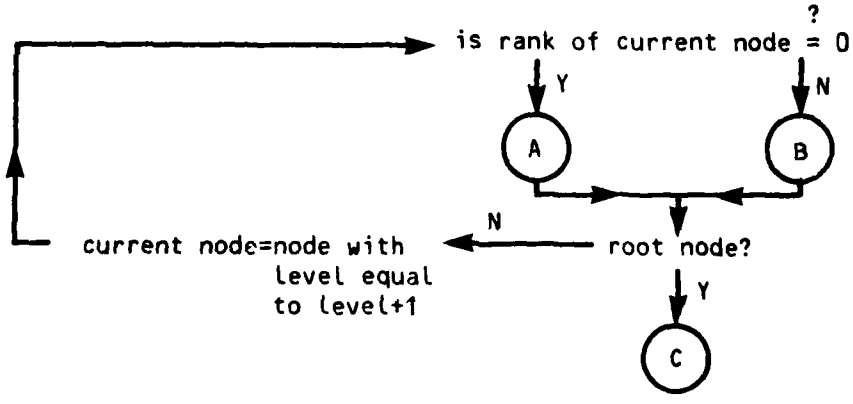


Figure 6.9: Flowchart for the level algorithm.

```
WHERE TREE.CURRENT=1 AND TREE.NODE#  =Ø DO
    WHERE (SELECT TREE.NODE#  = (SELECT CHILDREN.NODE#
           FROM TREE)            FROM CHILDREN) DO

              A

    ELSEWHERE

              B

    UPDATE TREE
    SET TREE.CURRENT=1
    WHERE* TREE.LEVEL#+1 =
        SELECT UNIQUE TREE.LEVEL#
        FROM TREE
        WHERE* TREE.CURRENT=0
    UPDATE TREE
    SET TREE.CURRENT=Ø
    WHERE*
           (SELECT TREE.NODE#
            FROM TREE
            WHERE* TREE.CURRENT=1)
            INTERSECT
           (SELECT MIN(TREE.LEVEL#)
            FROM TREE)
ELSEWHERE

            C
```

Figure 6.10: The level algorithm program.

relation WHERE* either an attribute satisfies a specific condition or the attribute is contained within a specified set of values. The special operation GROUP BY, the HAVING clause, and the library function SET can be used in the retrieval operations. GROUP BY partitions the pertinent relation into groups such that within any group all tuples have the same value for the attribute specified following the GROUP BY operator. The HAVING clause is a special form of the WHERE* clause applying the groups. Entire groups are chosen or discarded based on the condition specified in the HAVING clause [Dat]. The SET function returns the set of values occurring for the specified attribute within a given group.

The storage operations used in the two programs are UPDATE, INSERT, and DELETE. The UPDATE command will UPDATE a relation and SET the values for the attributes WHERE* certain conditions are satisfied. The INSERT and DELETE commands are self-explanatory.

Some of the library functions used excluding the SET function are COUNT and MIN.COUNT returns the number of occurrences of the specified attribute. When COUNT is used in conjunction with a GROUP BY operator, a COUNT is returned for each group. MIN returns the MIN (minimum) value of the specified attribute.

Some additional details about the proposed host language need to be given to clarify the reasons for the changes to be suggested. The focus of the suggested changes are index variables and their use in conditional statements. As currently defined, index variables must be scalars whose ranges are specified in a range list. In addition, Micro

Controller variables are supposed to be scalars, while PCU variables are supposed to be arrays.

It should be noted that there are two different types of WHERE commands used in the programs. One WHERE command is part of a WHERE-DO structure in the host language while the other WHERE command is a part of the relational language and denoted WHERE*. The command names will probably need to be changed to reduce confusion and simplify command processing.

Since relations are essentially two dimensional arrays they would be considered to be PCU variables. The SEQUEL commands choose subsets of the relations much as the index variables are used to specify index sets. The SEQUEL commands were therefore used to specify index sets in the conditional PCU processor control statements. The major difference between the use of the index variables in specifying an index set and the use of SEQUEL commands to choose an index set is the inability to specify a range list using the SEQUEL commands. The SEQUEL specified set is always dynamic and the values of the indices are always implicit.

The organization of the SEQUEL commands in the program in Figure 6.9 is impractical in terms of compilation. A more practical approach would be to use index variables to store the values of the index set specified by the SEQUEL commands. For example, consider the first condition in the first WHERE statement:

```
SELECT GRAM.TERM    =    SELECT TREE.SYMB
FROM GRAM                FROM TREE
WHERE GRAM.RANK=0        WHERE TREE.CURRENT=1
```

Let G and T be defined as index variables in the declaration statements

at the beginning of the program. The following initialization statements would be needed before the control statement.

```
G  ←  SELECT *
      FROM GRAM
      WHERE* GRAM.RANK=0
T  ←  SELECT *
      FROM TREE
      WHERE* TREE.CURRENT=1
```

An * refers to tuples in a relation. This allows specification of the desired attribute in the WHERE statement which adds to the clarity of the program. The WHERE statement would then read:

WHERE GRAM.TERM [G] = TREE.SYMB [T].

At points in the program there are references to relations without indices, only the relation and attribute names are given. This is the standard form of reference to a relation. In a relational language such as SEQUEL, there is no provision for the specification of indices. The choice of the index subset is supposed to be transparent to the user. Since using the proposed PASM parallel language commands requires a knowledge of index variables and their use on the part of the user, it did not seem unreasonable to remove at least a part of this transparency. As a result, the programmer will have a little more flexibility in manipulating the relations.

While allowing the programmer to specify indices, it is not necessary to require that indices be used at all times. However, as was mentioned previously, for practicality the programmer will be required to use indices in certain conditional statements.

A suggestion for a means of allowing both ways of specifying subsets of a relation would be to have automatic index variables, one

AD-A113 934 PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING F/G 5/8
THE USE OF DATABASE TECHNIQUES IN THE IMPLEMENTATION OF A SYNTA--ETC(U)
DEC 81 E C SEED, H J SIEGEL AFOSR-78-3581
UNCLASSIFIED TR-EE-81-49 AFOSR-TR-82-0301 NL

2 OF 2
AD A 113934

index variable per relation. Programmer defined index variables could easily be grouped together during compilation to be synonymous with the appropriate automatic index variable. The automatic index variable could be handled in a manner analogous to the handling of nested WHERE statements. The user also would have the ability to access individual tuples; an index value can be used in place of an index variable.

Although allowing the user to specify index variables and values is a departure from the conventions of relational database processing, it provides an interface between a strict relational language and the proposed PASM parallel language commands. Theoretically the user should be able to take full advantage of the relational database techniques, or to mix relational commands with PASM parallel language commands as was done in the programs in Figures 6.9 and 6.11, or to manipulate the relations utilizing only the proposed PASM parallel language commands.

Some of the SEQUEL commands can be preprocessed to yield PASM parallel language commands. For example, simple SELECTs can be translated into WHERE statements simply by extracting the WHERE* statement of the SELECT-FROM-WHERE* group.

In contrast, some of the more complex SEQUEL commands are not so easily translated. Efficient means of implementing some of the set operators and commands such as the GROUP BY will need to be found. It is through the use of commands such as these that the programs can be greatly simplified, but without relatively efficient code the ultimate goal of significantly reducing the processing time cannot be accomplished. Some ideas for the implementation of these commands will be discussed in the next chapter.

Another suggested change to the proposed PASM parallel language commands is to allow simple one-dimensional arrays containing counts or indices to be allowed as Micro Controller variables. This was useful in the programs when forming the triplet combinations. An array was needed because of the multiple values returned when a SELECT-FROM-GROUP BY group of commands was used with a COUNT in the SELECT. This type of command is quite useful for returning multiple counts of items which in turn may conceivably be needed in a Micro Controller control statement.

This chapter has presented the relational database as a feasible tool for the implementation of the relatively complex structures required by a syntactic pattern recognition task. Some suggestions for embedding relational database language commands and schemas in a program written in the parallel language proposed for PASM were made. Some proposals for the implementation of these relational commands and for the relational database itself will be made in the next chapter.

CHAPTER 7

PROCESSOR AND DATA ALLOCATION FOR THE RELATIONAL APPROACH

7.1 Introduction

Consider the level traversal algorithm for purposes of discussion. The processor and data allocation problems posed by a relational task differ from those posed by image processing tasks even though relations are somewhat similar to the arrays encountered in image processing. Often the index sets of an image processing task are predefined and well ordered making it possible to determine processor and data allocation schemes at the time of compilation. In contrast, in a relational database task the index set will be determined during execution rather than at compile time. Also, an image processing task usually uses a parallel machine to reduce the number of calculations to be performed while a relational task uses a parallel machine to decrease the time taken to perform searches and comparisons. Thus it is evident that a relational task poses several new processor and data allocation problems that need to be considered.

The dynamic index sets pose the problem of how to allocate the processors and distribute the data such that as many members of the index sets as possible will be processed in parallel. One case will be discussed to further clarify this problem. In this case, blocks of tuples in the relations will be distributed to a limited number of

processors. In this instance, all of the 7x7 windows of a 98x98 picture are being processed in parallel.

The suggestions for processor and data allocation are made assuming the whole picture is represented by multiple databases, one database per window. An alternative to the multiple database approach might be as follows. The relations as diagrammed in Figure 6.6 will be altered so that the whole picture is represented in the relations. To accomplish this, an additional attribute, the window attribute, will need to be added to all of the relations except the GRAM relation. This implies that the windows would need to be numbered and that these numbers will be the values of the window attribute. The processor and data allocation schemes possible for this approach are relatively simple extensions of those described in the following sections.

7.2 Processor and Data Allocation

The distribution of the tuples of the CHILDREN, TREE, and GRAM relations, among the processors for an individual window and for a picture will now be considered. For the sake of simplicity, it is initially assumed that the user is specifying the processor and data allocation. This section will describe the distribution of blocks of tuples of a relation to the available processors. For example, the TREE relation is assumed to be sorted by node number. The TREE relation tuples would then be distributed to the processors in rank groups. It should be noted that this assumes the rank of the node is readily available. The relations given in Figure 6.6 do not include the RANK attribute. An additional relation would have to be added with the attributes NODE# and RANK to provide this information. The other

alterntive would be to determine the rank during the loading of the database of determining the number of children of a node. This, however, might become a fairly time consuming process. One processor will receive the nodes of the same rank. Of course, this depends on the number of different ranks there are and how many processors are available for a window. A single processor may actually contain a subset of the set of nodes of a given rank if there are a large number of processors per window or it may contain multiple sets of nodes with different ranks if there are only a few processors per window. The CHILDREN relation would be distributed in the same fashion as would the GRAM relation. The grammar rules are also assumed to be sorted by rank. The grammar rules would be distributed to the processors such that the rank of the nodes stored in a single processor is equal to the rank of the grammar rules in that processor.

In this instance, the data in the GRAM relation is distributed according to the value of the primary sort key attribute, the value of the rank. Determining the number of processors to allocate to each window database is simple once the total number of window databases is known. The programmer will have to tell the compiler how many window databases will be processed. This statement is probably best placed at the beginning of the program and could be as simple as WINDOWS=n, where the program is considered to describe the processing for one window.

As an example, consider the 1024 processor machine, 98x98 picture, and 7x7 window used in Chapter 5. Since there are 196 7x7 windows with only one window per group of processors, there would be approximately five processors per window. In order to enable PASM to process all the

windows at the same time and to allow for interconnection network transfers, there would have to be four processors per window, where four is the nearest power of two less than five. Conveniently for this example, there are four processors and four different rank values so that there is a one-to-one correspondence between the sets of nodes and grammar rules sorted by rank and the number of processors. In this example, there would be no data transfers between processors of rank 1. All of the processing for nodes of a rank 1 would occur in a particular processor. It is suggested that the processing of the nodes of rank 2 and the nodes of rank 3 be distributed across the processors responsible for storing the pertinent tuples for these nodes. In this way there should be some time savings in the processing of the triplet combinations for those nodes.

A fair percentage of the processors should be active using this allocation and processing scheme for the level algorithm program. The level algorithm spreads the processing of the lower rank nodes over several serial steps. Only two out of each of the 2W size sets of lower rank nodes are processed at the same time with a maximum of W-3 nodes being processed together in any one processor.

7.3 Mode of Processing

In the example discussed, all processing in one processor was assumed to be performed independently. Each processor was performing a separate task involving different amounts of data and different code. It would seem that IMD mode would be the best mode of processing for the level algorithm program. MIMD mode would also be recommended for the case where nodes of multiple ranks are in one processor. However,

for the case when enough processors are available so that the nodes of one rank are distributed across more than one processor, the mode would need to be MSIMD to fully take advantage of the parallel processing within a node group.

7.4 Compilation Problems

Some suggestions for resolving one of the compilation problems will be made in this section. The problem is how to translate the GROUP BY commands which require sorts as efficiently as possible. Data definition languages usually allow the user to state which relations or records are to be sorted, the sort key and the type of ordering desired. The RELATION data definition statements as suggested in the previous chapter do not include statements for specifying sorted data. It may be possible to determine some of the sort specifications from the GROUP BY commands in the programs because the GROUP BY commands implicitly require sorting. This would shift some of the burden of sorting requirements from the programmer to the preprocessor and compiler. This may also aid in the reduction of the number of run time sorts needed because it gives the machine the knowledge of what sorts will be necessary before execution thereby allowing the machine to presort certain relations. Static relations, for example, should be sorted during the database loading process.

At the expense of a certain amount of data redundancy, it may be reasonable to presort the static relations in all the different ways as required for execution of the GROUP BY commands in the program. This suggestion is made assuming that most of the relations used in syntactic pattern recognition tasks will require only one sort. For the leaf and

level algorithm the number of multiple sorts in a static relation depends on what initial sorting is done, but will be small.

The GROUP BY commands in the programs do not necessarily contain all the needed sort key attributes. It will be necessary in some instances to require the programmer to specify how the data is ordered. This may be accomplished by including a series of GROUP BY statements in the data definition statements. For example, to indicate that the TREE relation is sorted by NODE# the following statement might be used:

```
RELATION TREE
GROUP BY NODE#
```

This would have to follow the TREE relation declaration.

The GROUP BY statements can also be used for the dynamic relations. However, unlike the static relations, the dynamic relations will need to be sorted during processing. The GROUP BY statements for the dynamic relations might be used to generate code that would cause the tuples to be distributed to the appropriate processor as they are formed during processing.

From the above discussion, it would seem that the necessary information for the implementation of the allocation scheme discussed in section 7.2 can be obtained from the program. However, this allocation scheme is not necessarily a particularly effective scheme in terms of the speed of processing, but it was sufficient here to illustrate some of the problems that can be encountered in an allocation scheme. Further research needs to be performed in this area to determine how relations might be best distributed and processed on PASM.

CHAPTER 8

CONCLUSIONS

A tree parsing task was investigated to determine how PASM might process such a task. It was hoped that a parallel machine such as PASM would be able to significantly reduce the processing time for a syntactic pattern recognition task such as the tree parsing task. Although definitive results are not obtainable at this time, it appears that PASM could greatly reduce the processing time.

Initially, two approaches to the order of processing of the tree data, the leaf traversal and the level traversal, were investigated. Using the two tree parsing algorithms resulting from these two traversals, some tentative processor and data allocation schemes and processing modes were discussed for illustrative purposes. These schemes assumed a tree structure would be used for storing and processing the data. The discussions indicated that PASM's reconfigurability gives a wide variety of options for processor and data allocation and the mode of processing. The leaf traversal approach could make good use of both the SIMD and MIMD modes, while the level traversal may be better suited for MIMD implementations. With suitable analysis tools, this flexibility could be put to use to yield the best time savings.

The next factor considered was the implementation of the data structures and the algorithms. It was felt that the data structures were best represented by databases and that a mixture of database language commands and high level parallel processing commands for PASM could be used to implement the algorithms.

The first database model investigated was a CODASYL model. It was felt that this model would not be able to fully exploit PASM's parallel processing capabilities. The CODASYL model and DML were discarded in favor of the relational model and a relational language.

The use of a relational database and a relational language provide PASM with the ability to process the relatively complex data structures used in syntactic pattern recognition tasks. The simplicity of the relational approach in contrast to the detail required in the use of the network and hierarchical approaches makes the implementation of syntactic pattern recognition tasks more practical. The simplicity of the relations to basic image arrays is one of the major reasons for the apparent success of the relational approach. Relations and relational languages provide an interface between complex structures and arrays.

The allocation scheme for the level program was proposed primarily for illustrative purposes. Further research is needed to determine optimal allocation schemes for processing relations on PASM. This research would be able to take into account the findings of other researchers interested in processing relations on parallel machines. The results of this paper would seem to indicate that significant time savings may be possible through the use of relational techniques to process syntactic pattern recognition tasks on PASM.

LIST OF REFERENCES

LIST OF REFERENCES

[Bou] Bouknight, W. J., et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

[CF1] Chang, N. S. and Fu, K. S., A Study on Parallel Parsing of Tree Languages and Its Application to Syntactic Pattern Recognition, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-15, Mar. 1978.

[CF2] Chang, N. S. and Fu, K. S., "Parallel parsing of tree languages," 1978 IEEE Computer Society Conf. Pattern Recognition and Image Processing, May 1978, pp. 262-268.

[Dat] Date, C. J., An Introduction to Database Systems, Second Edition, Addison-Wesley, 1977.

[Fly] Flynn, M. J., "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

[Fu1] Fu, K. S., Syntactic Methods in Pattern Recognition, Academic Press, 1974.

[Fu2] Fu, K. S., "Tree languages and syntactic pattern recognition," in Pattern Recognition and Artificial Intelligence, ed. by C. H. Chen, Academic Press, 1977, pp. 257-291.

[FuB] Fu, K. S. and Bhargava, B. K., "Tree systems for syntactic pattern recognition," IEEE Trans. Comp., Vol. C-22, Dec. 1973, pp. 1087-1099.

[Ger] Gerritsen, R., SEED Reference Manual, International Data Base Systems, 1977.

[HaW] Haseman, W. D. and Whinston, A. B., Introduction to Data Management, Richard D. Irwin, 1977.

[IC1] IBM Corporation, Information Management System/Virtual Storage General Information Manual, IBM Form No. GH20-1260.

[IC2] IBM Corporation, Interactive Query Facility (IQF) for IMS Version 2, IBM Form No. GH20-1074.

[LF1] Lu, S. Y. and Fu, K. S., "Error-correcting tree automata for syntactic pattern recognition," IEEE Trans. Comp., Vol. C-27, Nov. 1978, pp. 121-126.

[LF2] Lu, S. Y. and Fu, K. S., "Structure-preserved error-Correcting tree automata for syntactic pattern recognition," 1976 IEEE Conf. Decision and Control, Dec. 1976.

[Mar] Martin, J., Principles of Data-Base Management, Prentice-Hall, 1976.

[MSS] Mueller, P. T., Jr., Siegel, L. J., Siegel, H. J., "A parallel language for image and speech processing," Proceedings of the IEEE Computer Society's Fourth International Computer Software and Applications Conference, COMPSAC 80, Oct. 1980, pp. 476-483.

[Nut] Nutt, G. J., "Microprocessor implementation of a parallel processor," 4th Symp. Comp. Arch., Mar. 1977, pp. 147-152.

[Si1] Siegel, H. J., "Analysis techniques for SIMD machine interconnection networks and the effect of processor address masks," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 153-161.

[Si2] Siegel, H. J., "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comp., Vol. C-28, Dec. 1979, pp. 907-917.

[Si3] Siegel, H. J., "The theory underlying the partitioning of permutation netwo[illegible]," IEEE Trans. Comp., Vol. C-29, Sept. 1980, pp. 791-801.

[SiM] Siegel, H. J., McMillen, R. J., Mueller, P. T., Jr., and Smith, S. D., A Versatile Parallel Image Processor: Some Hardware and Software Problems, School of Electrical Engineering, Purdue University, Technical Report TR-EE 78-43, Oct. 1978.

[SiS] Siegel, H. J. and Smith, S. D., "Study of multistage SIMD interconnection networks," 5th Symp. Comp. Arch., Apr. 1978, pp. 9-17.

[SmS] Smith, S. D. and Si[illegible] J., "Recirculating, pipelined, and multistage SIMD interconnection networks," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 206-214.

[SM1] Siegel, H. [illegible] and McMillen, R. J., "Using the augmented data manipulator network in PASM," Computer, Vol. 14, Feb. 1981, p. 25-33.

[SM2] Siegel, H. J., and McMillen, R. J., "The multistage cube: a versatile interconnection network," Computer, Vol. 14, Dec. 1981, pp. 65-76.

[SMS] Siegel, H. J., Mueller, P. T., Jr., and Smalley, H. E., Jr., "Control of a partitionable multimicroprocessor system," 1978 Int'l. Conf. Parallel Processing, Aug. 1978, pp. 9-17.

[SSK] Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., Smith, S. D., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Trans. Comp., Vol. C-30, Dec. 1981, pp. 934-947.

[WuB] Wulf, W. A. and Bell, C. G., "C.mmp - a multiminiprocessor," Proc. FJCC, Dec. 1972, pp. 765-777.


FILMED
5-8